Scala-Mungo

Final Report

MENG Individual Project ENG5041P

Alice Ravier - 2206245R

Electronics and Software engineering MEng student

Supervisors: Dr Ornela Dardha and Dr Nicholas Bailey

Department of Science and Engineering

University of Glasgow

Scotland

2020, 2021

Abstract

In programming languages nowadays, the concept of data types is prevalent and helpful in ensuring that code executes properly. Programmers will often be forced to pick a datatype for their variables and only allowed certain operations on them.

However, when a programmer wishes to create code which should behave according to a set of rules (a *protocol*), there is no way of ensuring that the protocol is well followed by assigning a datatype. Since inspection is a costly and often ineffective way of removing errors, an automated approach is desired, which is where *typestates* prove useful.

Typestates enable each class in a program to have a protocol attached to it. The program can then be checked at compile time for protocol violations.

This report presents Scala-Mungo, which is a tool which incorporates typestates into Scala. It is heavily inspired by Mungo, a similar tool for Java.

Scala-Mungo is the first typestate tool to implement *unrestricted aliasing*. This enables users to use aliases freely in their code, without losing the ability to check their code for protocol errors. The implementation relies on a global approach to tracking instances.

The results of user testing show that Scala-Mungo is user-friendly, intuitive, and mostly bug-free. However, some alterations and a more expansive evaluation should be undertaken before it is released to the public.

Acknowledgements

Thank you to my first supervisor Ornela Dardha for all of the help with the project and supporting me when I was having a rough time.

Thank you to my second supervisor Nicholas Bailey for the advice and fun anecdotes.

Thanks to Mathias Steen Jakobsen for helping me with fields and advice on the literature survey.

Thanks to João Mota for helping me understand his Mungo implementation, enthusiasm and very quick replies to my emails.

Thanks to Laura Voinea for helping me with the original Mungo.

A great thanks to my amazing girlfriend Sofia Simola for her unbelievable support throughout the project and her help with proofreading and sorting out my references.

Thanks to my great friend Piotrus Watson for always being up to help, chat or destress with some games; and the proofreading and great advice.

I really couldn't have done it without you guys :)

Thank you to StackOverFlow user Dmytro Mitin, and Scala Contributor SethTisue for answering my questions about Scala compiler plugins.

Special thanks to my user testers:

João Mota

Laura Voinea

Mathias Steen Jakobsen

Laroslav Golovanov

Contents

1 Introduction **1**

1.1 Scala-Mungo . . . 3

2 Literature survey **5**

3 MoSCoW Requirements **10**

4 Design **12**

4.1 General design concerns . . . 12

4.2 Protocol language . . . 12

4.3 Protocol checker . . . 15

4.4 Tying things together . . . 16

5 Implementation **18**

5.1 Protocol description language . . . 19

5.1.1 Interface code structure . . . 19

5.1.2 Processing data . . . 20

5.1.3 Testing . . . 22

5.2 Protocol checker . . . 22

5.2.1 Overview . . . 23

5.2.2 Fields . . . 27

5.2.3 Basic checking . . . 28

5.2.4 Control flow structures . . . 30

5.2.5 Aliasing and assignments . . . 37

5.2.6 Functions . . . 38

5.2.7 Return values . . . 43

5.2.8 Testing . . . 47

6 Case study **48**

7 User testing **55**

7.0.1 Aims . . . 55

7.0.2 Methods . . . 55

7.0.3 Results and discussion . . . 56

7.1 Conclusions of the user testing 57

8 Conclusion and future work **58**

8.1 Conclusion 58

8.2 Future work 58

8.2.1 Features which could be added 58

8.2.2 User testing changes 59

8.2.3 Further evaluation 59

8.3 Scalac phases 63

8.4 Aliasing examples 63

8.5 User testing 64

8.5.1 Scala-Mungo Tutorial 64

8.5.2 Questions 74

8.5.3 Answers 74

List of Figures

1 Graphical representation of a typestate attached to a simple email client. 2
2 The different components of the Scala-Mungo tool interacting together. 17
3 Illustration of the protocol for a class used as a running example in the **Implementation** section. 18
4 The classes involved in the implementation of the protocol language. 20
5 The main classes involved in the implementation of the checker. 23
6 Visual representation of aliasing an instance inside of the Main object. 25
7 Additional classes for the checker: Function and Element classes. 26
8 Visual representation of the trackedElements map 27
9 Graph of the MoneyStash protocol. 48
10 Graph of an excerpt of the SMTP protocol 52

List of Tables

1 Internal representation of a state machine as a two dimensional array of states 21

1 Introduction

In this section, we will start by explaining the problem Scala-Mungo seeks to solve; namely the lack of verification for protocols in Scala. Then, we will expand on different existing approaches to solve this problem. We will then talk about the challenge which aliasing poses and how our tool resolves it. Finally, we will consider in what ways Scala-Mungo contributes to the existing literature.

Types in computing lack verification for protocols

In programming languages nowadays, the concept of data types is prevalent and helpful in ensuring that code executes properly. Programmers will often be forced to pick a datatype for their variables and only allowed certain operations on them. If they violate the rules, the code will not compile, and therefore it will not be allowed to run and potentially cause damage.

However, when a programmer wishes to create code which should behave according to a set of rules (a *protocol*), there is no way of ensuring that the protocol is well followed by using a datatype. If a programmer wants to ensure that their email client only allows access to email after authentication; or if they want to ensure that an amount of money is never used before interest is applied to it, they must either hope they have written their method calls in the correct order, or add state to their classes. In the latter case, they could add an "authenticated" boolean variable to the email client to add state to it. This still does not remove their burden of manually checking that their method calls properly update the state of the email client.
Since inspection is a costly and often ineffective way of removing errors, an automated approach is desired, which is where *typestates* prove useful.

Typestates

A typestate is a concept in computer science meant to address the above issue, first formalised in Strom and Yemini (1986). Attached to a class, it indicates which state the class currently is in and which methods are available to it in its current state.
It can be thought of as enforcing a protocol, or describing the valid sequences of methods a class instance can do.
It can also be represented as a graph. Below is a simple typestate for an email client represented as a graph:

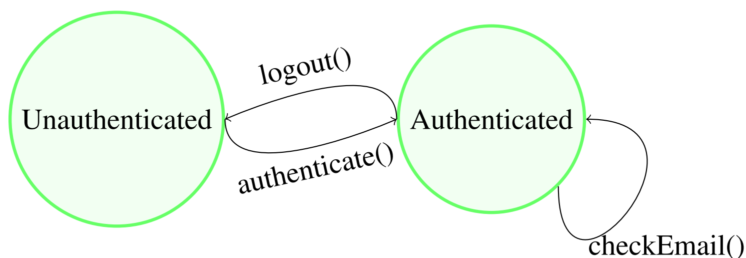


Figure 1: Graphical representation of a typestate attached to a simple email client. The nodes represent states, and the arrows transitions. The methods facilitating the transitions are written next to the arrows.

Given the typestate in figure 1, the email client now has a concept of when the "checkEmail()" method is allowed; i.e. only in the *authenticated* state.

The change from one state to another, through a method call, is called a *transition*. In figure 1, there are three defined transitions represented by arrows.

Essentially, a typestate describes the possible transitions a class can make.

We will now look at existing approaches to integrating typestates into programming languages, and compare them to our approach for Scala with Scala-Mungo.

Existing approaches to typestates

The current body of work on typestates generally involves creating a new language which incorporates typestates (Strom and Yemini (1986), DeLine and Fähndrich (2004), Saini et al. (2010), Coblenz et al. (2020)). This is not often practical since it requires the user to learn a new language and incorporate it into their codebase. Additionally, the languages require the user to incorporate a lot of typestate descriptions into their production code. This imposes extra work on the user and clutters the code.

An alternative to creating a new language is to write a tool which affixes itself into an existing language. This is what Mungo (Kouzapas et al. (2018)) does with Java, and nitnelave (2019) does with C++.

In nitnelave (2019), the user creates wrappers for their classes which enforce protocols. In Mungo, the user defines their typestate in a separate file, attaches it to their class and can then code as usual in Java.

1.1 Scala-Mungo

The work presented in this report aims to investigate typestates in Scala. Scala-Mungo is, to our knowledge, the first implementation of typestates in Scala. It is heavily inspired by Mungo and adds typestates to Scala in the same modular way. This means that no language extensions to Scala are necessary.

There is one way in which Mungo and Scala-Mungo considerably differ: aliasing.

The challenge of aliasing

Aliasing presents itself as a challenge for all approaches to typestates.

Aliasing, or the act of giving one object in memory different names, is an important issue which must be dealt with to avoid losing track of an object's current typestate. In this example, we can see what might happen if we do not keep track of the aliases of an object.

The typestate on the Cat class enforces that the walk() method can only be used once.

```
1 var x = new Cat
2 var y = new Cat
3 x=y
4 x.walk()
5 y.walk()
```

As walk() is only called once on x and y respectively, no error will be thrown. However, in terms of the Cat object, the method walk is called twice, which should cause a protocol violated error.

Other typestate implementations either disallow aliasing (Strom and Yemini (1986), Kouzapas et al. (2018)) or enable it with restrictions, usually only letting aliased instances be read, rather than letting them call methods.

Scala-Mungo proposes an approach which allows *unrestricted aliasing* of instances. This permits users to alias instances just as they would in a normal program while still getting the benefits of typestate analysis.

Contributions

To our knowledge, Scala-Mungo is the first tool which offers the following items.

- **Typestate analysis in Scala.** This enables automated protocol checking. The design and tooling is discussed in section 4, and the implementation is described in section 5.
- **An intuitive, Scala-like interface for defining protocols.** The details of the interface are described in section 4. It aims to be concise and similar to other Scala tools.

- **Unrestricted aliasing.** This is expanded on in sections 2 and 5. It allows the user to alias variables freely.

- **No need for language extensions.** No modifications to the Scala language were made to enable this tool.

Scala-Mungo is part of a paper to be published in an international conference in Spring 2020 (Ornela Dardha (2021)). The source code for Scala-Mungo can be found on the github repository at: https://github.com/Aliceravier/Scala-Mungo (Aliceravier (2021)).

Report structure

This report examines the *current body of work* on typestates and aliasing in section 2.
The requirements for Scala-Mungo are established as *MoSCoW requirements* in section 3.
The *design* to meet the requirements is explained is section 4, followed by an in depth view of the *implementation* in section 5.
A *case study* is described in section 6, to highlight the functionality of Scala-Mungo, followed by the results of *user testing* in section 7.
Finally, in section 8, the *conclusion* summarises the report and possible *future work* on the project is discussed.

Notation

The notation used in this report is as follows:

- code is is the monospaced font, size 10pt. Keywords are in green, strings are in blue, comments are in orange, and the rest of the code is in black.

- citations are in green, section and figure references are in blue, and urls are in teal.

2 Literature survey

In this literature survey we will start by outlining the different approaches to verifying protocols in programming languages. Then we will summarise the existing literature on the *typestate* approach used in this project. Then, we will describe different approaches to *aliasing* in typestate-based tools, as well as the existing approaches to protocol checking in Scala.

Approaches to verification of protocols

Types that describe how a computation should proceed are referred to as *behavioural types* (Hüttel et al. (2016)). S. Gay and Ravara (2017) presents uses of behavioural types, such as to ensure that APIs are behaving in the way they claim to, or to check the validity of concurrent systems.

One such behavioural type is the *session type*.

First formalised in Honda et al. (1998), Session types focus on the allowed communication in a program (Vallecillo et al. (2006)). They are derived from pi-calculus (Sangiorgi and Walker (2001), Dardha et al. (2012)), which is used to model concurrent systems. A program with multiple communicating components (or parties), can be described with session types. A session channel describes the series of communication allowed between parties. It can be split into different endpoints, each owned by a different party. The endpoints define what inputs and outputs the party is allowed at any given time in the program.

Session types can apply to two parties communicating, or more with multiparty session types (Scalas et al. (2017)). They have been integrated into many programming paradigms, such as object-oriented languages (S. J. Gay et al. (2010)), and functional languages (S. Gay and Vasconcelos (2007)).

Another behavioural type is the *choreography*. Choreographies stem from the idea that programmers should write their communicating programs as a list of communications, rather than implement different communicating endpoints separately (Montesi and Yoshida (2013)).

For example, see the following choreography (the example is inspired from Montesi (2014) and the syntax is derived from "Alice and Bob" notation, presented in Needham and Schroeder (1978)) :

```
1 1: petrolStation -> carOwner: petrol;
2 2: carOwner -> petrolStation: money
```

Here, the petrol station gives the car owner petrol, after which the car owner gives the petrol station money.

This choreography can then be expanded into implementations for the petrol station and the car owner.

Choreographies have been proven to be Turing complete (Cruz-Filipe and Montesi (2020)) and intrinsically

do not contain deadlocks (Cruz-Filipe et al. (2019)). The main difference between choreographies and session types is that choreographies start with the communication between objects and construct the implementation; whereas the session types start with the implementation and create the types (Cruz-Filipe et al. (2019)).

Session types and Choreagraphies focus on communication between processes. In an object-oriented context it is more natural to model the *state* of objects rather than communication patterns. Therefore, it was determined that *typestates* would work better for Scala-Mungo. We will discuss this typestate approach in the next section.

Typestates

Typestates let one associate a type representing the state of a protocol to a class. It defines the set of methods allowed by a class at a certain point in the program and can change over the course of a program.

Strom and Yemini (1986) first introduced the concept of typestates. They put it forward as a way of automatically detecting errors in protocol at compile time. They implemented the toy language *NIL*, which was built as a prototype for systems software. Their implementation is based on imperative programming and their typestates are therefore not linked to objects but to variables. Correspondingly, their typestates ensure correct initialisation rather than correct method calls.

DeLine and Fähndrich (2004) built on this idea by creating *Fugue*, a object oriented language which heavily relies on the programmer to add protocol specifications to their classes. Pre- and post- conditions must be added to all methods explaining how the parameters change states, and how they change the state of the class. The checker then goes through the program making sure the method order makes sense, and verifies no null pointers are dereferenced, i.e. that no variables are null when being accessed.

Saini et al. (2010) presents *Plaid*, another protocol checking language. In this language, the programmer defines a class as a series of states, each with its allowed methods. In the methods, the possible state change is also defined. Once again, this puts impetuous on the programmer to understand how each of their methods influences the states of the object and the parameters.

Mungo, presented in Kouzapas et al. (2018), is much less demanding of the user and only asks that they write a specification file with their protocol, as opposed to changing all their method definitions. It is a tool extending Java rather than replacing it with a new language. It also introduces the *"end"* state which enables it to check if classes have reached completion of their protocol. This is the approach used for Scala-Mungo, which is heavily

inspired by Mungo.

nitnelave (2019) is a C++ library which implements typestates. It lets users define their protocol in a subset of C++. Users then create a class with a protocol by creating a wrapper which contains the class and the protocol.

Coblenz et al. (2020) introduce *Obsidian*, which uses typestates to create a blockchain language which can verify that transactions are done safely. Obsidian requires the user to write their transitions inside method definitions, similarly to Plaid.

Aliasing

We discussed aliasing in the introduction (section 1) with the example about the Cat class. Now, we will introduce different ways languages and tools have approached aliasing.

Some tools severely restrict aliasing. In Strom and Yemini (1986), the NIL language does not support pointers, which makes aliasing impossible and removes the problem.
Though supporting pointers, Mungo enforces *linearity* in programs, a concept which only allows one reference of an object to own the typestate at a time. When an alias is created, it is the only reference which is allowed to use the typestate. If the alias is assigned to another variable, that variable becomes the only one which can use the typestate. This eliminates aliasing problems by disallowing multiple simultaneous owners of a reference.

Since aliasing is an important part of programming, further research has been done about how to incorporate it into protocol checking tools and languages.

Clarke et al. (1998) offer a model which restricts aliasing to reduce problems with it. In their model, objects are given *contexts* beyond which they cannot be aliased. So if for example a car has an engine object, the engine cannot be aliased outside the car context.

In Fahndrich and DeLine (2002), the concepts of *adoption* and *focus* were introduced in the language *Vault* to deal with aliasing in a controlled manner. All objects start as being *linear* and therefore have mutable typestates. Adoption lets one alias an object and the created aliases are *nonlinear*, which means you cannot mutate their typestate.
Focus is a way to let one mutate the typestate of a nonlinear alias given certain conditions:

- the other aliases of the object must not be present in the focus block, and

- the typestate of the alias must be the same before and after the focus block

In Fugue, the concept of adoption is reused to let aliasing occur. An object can be either *NotAliased*, in which case it is known to have only one reference to it, or *MayBeAliased*, in which case it may have any number of references to it. Fugue only allows the typestate of a *NotAliased* object to change.

Bierhoff and Aldrich (2007) are the first to introduce the concept of *access permissions*. Access permissions determine whether an object can be accessed to change its typestate or only to read its value. It lets variable have different labels according to what permissions it and other aliases have to the object it is pointing to.
The variables are labelled as follows:

- They have the *unique* label if they hold the only existing alias to an object.
- They have the *full* label if they are the only alias which can edit the typestate, while other aliases can only read the object's state.
- They have the *immutable* label if they have read-only access to an object while all other aliases also only have read access to the object.
- They have the *share* label if all aliases have read and write access to an object.
- They have the *pure* label if they only have read access to the object but other aliases might modify it.

When an alias is created the access permission to the new references of the object are updated and generally become less permissive. The new references are given a fraction of the *unique* permission. If an alias is removed (it stops referring to an object in memory), the others can regain permissions.
Plaid uses access permissions to deal with aliasing.

Militão et al. (2010) build on access permissions by creating the concept of *views*. A view is a way of partitioning a class, and contains a number of its fields. This enables aliasing according to certain rules:

- An object can be aliased along its views as long as they do not intersect (so two aliases of a class instance can be created which access two different fields)
- An object can be aliased any number of times on a view which is *"unbounded"* (marked with a "!" by the programmer), but the aliases will only have read access to the object and cannot change its typestate. In this case, a fractional value is given to each alias (e.g. 1/2 and 1/2 for two aliases) which can be merged together to be of value 1, in which case the alias regains write access.

Coblenz et al. (2020) deals with aliasing by letting references be in different *modes*: *Owned*, *Unowned* and *Shared*. *Owned* means that the reference is unique. *Unowned* means there are multiple references to the object and one of them is *owned*. *Shared* means there are multiple references to the object and they are all *shared*. It only lets typestates be mutated from *owned* and *shared* references and enforces that only one *shared* reference mutate the typestate at once in a block of code.

All of the approaches which deal with aliasing require that the user become familiar with new concepts about aliasing, and often require them to identify exactly how each of their methods influence the typestate. This lets them approach typechecking in a modular way, that is, they are able to check small portions of the code without needing to know the rest of it.

Scala-Mungo seeks to lessen the cost to the user by not imposing that they write their code differently, at the expense of using a global method to deal with aliasing. In a global method, the entire program is analysed while keeping track of all the aliases.

Behavioural types in Scala

While Scala-Mungo is the first piece of work incorporating typestates into Scala, there have been implementations of session types into it before.

Scalas and Yoshida (2016a) adapts the notion of session types into Scala through a library called *lChannels*, Scalas and Yoshida (2016b). It enables a user to write a protocol and classes that follow it using its API. These classes can then be checked for correctness at compile time. It enables communication between two parties. This was later extended to enable more than two parties and integrated with *Scribble* (Yoshida et al. (2013)), a high level protocol specification language, in Scalas et al. (2017).

In summary, there have been many implementations of typestates into object oriented languages before. Scala-Mungo takes its inspiration from Mungo and lets the user use Scala normally after adding a protocol specification file to their program. In terms of aliasing, Scala-Mungo proposes a new, global approach with no restrictions on aliasing.

3 MoSCoW Requirements

For this project, two main tools were to be implemented:

- one permitting the user to input their protocol specification (henceforth called “protocol language”), and
- one which can check the protocol is being used correctly in the user’s code (henceforth called “checker”).

Here are the MoSCoW requirements for both of them (implemented items are highlighted in yellow):

Protocol language requirements

The protocol language:

Must:

- let the user describe their protocol (name states, define transitions)
- gather the protocol information into a useful data structure for the checker

Should:

- let the protocol branch based on what was returned from a method (for example, a different transition should be possible for each of the ”true” and ”false” values returned by a method which returns a boolean value)
- have an intuitive interface
- make checks about the validity of the protocol (for example, multiple states should not have the same name)

Could:

- have auto-completion of code
- automatically add in missing parts of the protocol if they are obvious, while throwing a warning to indicate the change to the user

Checker requirements

The checker:

Must:

- activate automatically when the user compiles the program

- check the validity of a sequence of method calls
- check methods correctly in the presence of common Scala constructs (if/else, match, loops, functions)

Should:

- check protocols attached to both classes and objects
- check fields of classes and objects which have protocols
- not require the user to write code which is interleaved with the code they would usually write. i.e. the protocol code should be separate
- check function calls in the user code with duplicate parameters
- traverse break statements in the user code
- be capable of processing multiple files of user code
- process try-catch-finally statements
- give useful error messages
- be thoroughly tested with automated tests

Could:

- be built online and easy for others to import into their projects
- have a useful tutorial which enables inexperienced people to use the tool
- support threading
- support collections of classes and objects with protocols
- support traits

During this project, all of the must and should haves were implemented, and some of the could haves. Additional desirable features are discussed in the future work section (section 8).

4 Design

This section will first present general design requirements for the whole project and then present the design of both the protocol language and checker parts. Figure 4.4 gives an overview of the different parts of Scala-Mungo and how they fit together.

4.1 General design concerns

Implementation language

A choice had to be made of what language to code in. As I am more familiar with Java, it initially seemed sensible to code in it.

However, it quickly became apparent that understanding Scala and how people code in it was a necessary skill for the project. Additionally, Scala is a concise language with type inference and syntactic sugar which makes it more pleasant to code in and easier to debug.

Therefore, Scala was chosen over Java despite the cost of slower coding at the beginning while getting used to the language.

Design aims

The set of design aims guiding this project are as follows:

- Create an accessible tool which is intuitive to use
- Make the interface as Scala-like as possible (i.e. concise and similar to other Scala tools)
- Create as little disturbance to the user as possible
- Make a tool up to the standards of the Scala community

4.2 Protocol language

In-line with the design aims, the Scala Developers forum ("Scala Contributors" (2018)) was consulted to determine what would be the best way to add a protocol specification to a Scala program. Creating a domain specific language (DSL) on top of Scala was the overwhelming response to this question. Since the Mungo tool dealt with this problem using Antlr (Parr (2012)), a tool which can parse and interpret a user made language, the two approaches are compared below.

Antlr VS a DSL

Antlr lets one create a language which it can then compile and from which data can be extracted. Mungo uses this to its benefit to let users write their protocols in plain JSON files, completely separate from the main program. Using Antlr for Scala-Mungo would enable protocols to be easily transferable between the two Mungos and would have a lower upfront cost since I am already familiar with Antlr.

However, Scala is very good for building domain specific languages (DSLs) and there are certain advantages to this approach over Antlr: Upon writing the protocol, users would get editor highlighting and thus immediate feedback if they were using the wrong syntax. This is not guaranteed for Antlr and requires installing the Antlr plugin.

Using a DSL also meant the different technologies of Antlr and Scala did not need to be interfaced. Therefore, everything could be written in Scala which made things faster and less complex to develop, as well as more intuitive to the user. Furthermore, it was determined that transforming Mungo protocols into Scala-Mungo protocols written in the DSL could be easily achieved with a simple script.

Thus, the choice was made to build the protocol language as a DSL.

Language design

To make the language as Scala-like as possible, it was designed to be concise and similar to other Scala DSLs such as "ScalaTest" (2009). ScalaTest reads almost like plain english, as Scala can let single argument methods be written without dots and parenthesis. Example from ScalaTest:

```
1 "A Stack" should "pop values in last-in-first-out order" in {
2     val stack = new Stack[Int]
3     stack.push(1)
4     stack.push(2)
5     stack.pop() should be (2)
6     stack.pop() should be (1)
7   }
```

This doesn't remove all programming constructs but makes it more concise and readable.

Based on this approach the protocol language looks like this:

```
object CatProtocol extends ProtocolLang with App{
    in ("init")
    when ("walk()") goto "State1"
    in ("State1")
    when ("comeAlive()") goto "end"
    in ("end")
    when("walk()") goto
        "init" at "true" or
        "end" at "false"
    end()
}
```

Sadly it was impossible to get rid of the parenthesis around the first method call in each line in the given project scope.

The functionality of the language is as follows:

- The user must write an object which extends ProtocolLang and put the code in the main function (done here by extending the object with `App`).
- They can define states with `in("stateName")`
- Methods possible in a state can be added underneath a state definition with:

  `when("method signature")goto "nextState"`.
- All methods defined in a protocol must be defined in the class(es) it is attached to.
- The user may want the state to go to to be dependant on the return value of the method and this is supported too, with the `at` keyword. The user can write

  ```
  when("method signature") goto
      "nextState" at "someReturnValue" or
      "anotherNextState" at "anotherReturnValue" or
      ...
  ```

  and can thus add as many different next states as they want as long as there are an equal number of different return values.
- This code is written on top of regular code and can thus contain things like comments or for loops which can aid in the creation of a protocol.
- Protocols must contain a unique `"init"` state which will be the state given to an instance when it is initialised.

- Protocols must contain a unique `"end"` state which indicates protocol completion, that is, at the end of a program, the object should be in the `"end"` state.
- Protocols must have an `end()` statement at the point the protocol is finished.

4.3 Protocol checker

The checker verifies that the protocol is followed correctly in the user program. It was decided this should be done at compile time to give the maximum amount of error correction before the program is run. This requires the checker to have some access to the user's code. Mungo uses a framework which allows it to add a type system into Java and analyse the program for protocol violations. Frameworks, and other potentially useful tools for this project, such as macros and compiler plugins, are examined and compared in the following section.

Macros VS compilerplugin VS Framework

Macros work with annotations the user would write and which are expanded into additional code during compilation. The problem with this is that it relies on the user to write annotations at the correct place for the code to be checked properly. This would put too much impetus on the user and not be in line with our design aims.

A compiler plugin inserts itself into the phases of compilation and can perform changes or additional checks to the code.

We decided to write a compiler plugin because this allows direct access to the code and gives the developer tools to work with it. Notably, it lets us access the abstract syntax tree (AST) of the code, which is a convenient datastructure through which to investigate the code. It also lets us match patterns to recognise certain structures in the code (for example an if statement).

Some frameworks are built as compiler plugins and let developers add functionality at a slightly higher level. However, none were found which would enable more code analysis than writing a raw compiler plugin and thus that was chosen for more flexibility and less overhead. Additionally, it was deemed that working on a framework with little support might cause significant problems if an issue came up.

A framework might have come in useful at the time of building the project since WartRemover (wartremover (2019)), for example, would let us add a component to their plugin easily. However, it was found preferable to keep our own plugin as a standalone so users do not need to download additional plugins.

Phase at which to add the plugin

The plugin must be placed to happen somewhere along the compilation process. These are grossly divided into four phases (for a full list of phases see the appendix, section 8.3):

- creating the abstract syntax tree(AST) and finding element types
- desugaring Scala's syntax into less concise code
- optimizing the code
- transforming the code into bytecode

The plugin was placed after the desugaring phase and before the optimisation phase since this gives us the most code to work with, which has element types and an AST. Placing it after the optimisation would risk missing errors or warnings for the code as some of it would have been optimised away.

4.4 Tying things together

Once the user has written their protocol in the protocol language, it must be tied to two other elements:

- the class the user wants the protocol to be attached to
- the protocol checker, so that it can check the protocol is being used correctly

To tie the protocol to a class, the same approach as in Mungo was used where the user can add a @Typestate annotation to bind a typestate file to a class. This approach was chosen because of its simplicity and so Mungo users did not need to learn a new way to add in their protocols.
Getting the protocol through to the checker is more complicated and involves finding the @Typestate annotation and the relevant data from the typestate file.
There are two main approaches to this problem:

- the checker could find the file and compile it, which would create a file with serialised data it could then look up.
- the user could compile their protocol themselves, which would then create the serialised data in a predetermined location, and the checker would just look up that location directly and retrieve the data.

In both cases, the data should be serialised while passing between the language and checker so that the checker can easily retrieve it and no explicit file parsing needs to happen. To achieve this, the native Scala serialization library was used.

In the first possibility, the user would need to wait twice as long for their program to compile every time they ran it, since two different programs must be compiled: the protocol and the user's program which uses it. The second approach lets the user only need to compile their protocol once and then use it as many times as they want without needing to re-compile it. It does mean they need to remember to compile it each time they create or update a protocol. The second approach was thought to be less frustrating for the user, because less time would be spent waiting, and was thus selected. Here is a diagram illustrating how the different parts fit together:

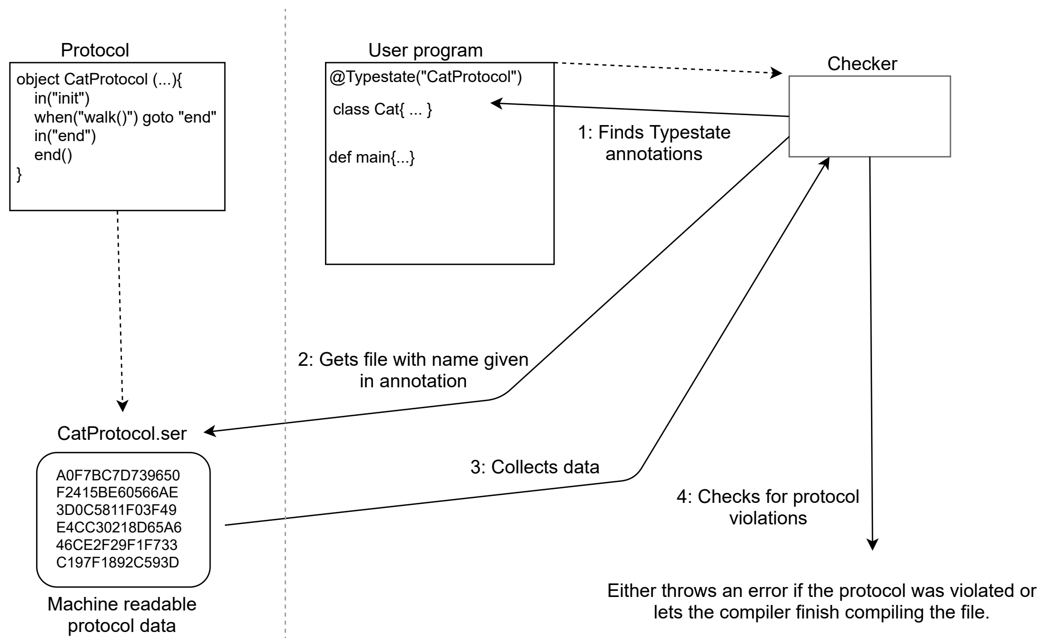


Figure 2: The different components of the Scala-Mungo tool interacting together. The dotted arrows indicate compilation. The user first creates their CatProtocol protocol and compiles it, creating the machine readable protocol data. Then the user writes their code using the @Typestate annotation and compiles it. During compilation, the checker fetches the compiled data and uses it to check the protocol is being followed correctly. The steps followed by the checker are numbered in the figure.

5 Implementation

In this section, we will go over the implementation of the protocol language and of the checker. To illustrate different features, the protocol in figure 3 will be used.

The nodes represent states in the protocol and the arrows represent transitions between the states. Methods facilitating the transitions are next to the arrows.

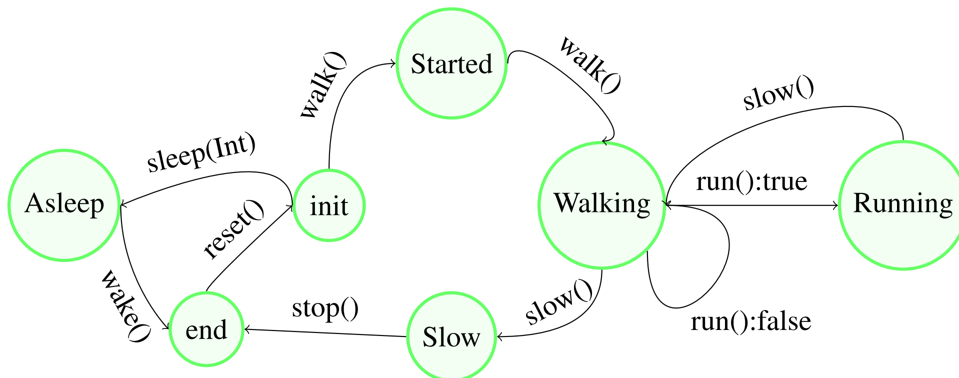


Figure 3: Illustration of the protocol for a class used as a running example in the **Implementation** section. The nodes represent states and the arrows represent transitions. The method names facilitating the transitions are written next to the arrows.

Translated into the protocol language this becomes:

```
in ("init")
when ("walk()") goto "Starting"
when("sleep(Int)") goto "Sleeping"
in ("Started")
when ("walk()") goto "Walking"
in ("Walking")
when("run()") goto
"Running" at "true" or
"Walking" at "false"
when("slow()") goto "Slow"
in("Slow")
when("stop()") goto "end"
in("Sleeping")
when("wake()") goto "end"
in("Running")
when("slow()") goto "Walking"
in("end")
when("startOver") goto "init"
end()
```

(See part 4.2 for an explanation of how the language works)

It will be assumed to be attached to the Cat class as such (assuming the protocol above is written in the file "CatProtocol.scala"):

```
1 @Typestate("CatProtocol")
2 class Cat{
3     def walk()={...}
4     def sleep(nb:Int)={...}
5     def wake()={...}
6     def stop()={...}
7     def run():Boolean={...}
8     def slow()={...}
9 }
```

5.1 Protocol description language

The protocol language is implemented as a domain specific langauge(DSL) on top of Scala. The implementation is divided into two parts:

- Interfacing with the protocol language, i.e. defining the commands used to define a protocol, such as `in("init")`.
- Processing the protocol definition into an appropriate format to send to the checker.

The structure of the interface will be described first, followed by a description of how the protocol is processed.

5.1.1 Interface code structure

The technique used to create the DSL interface was inspired from an online source: "Rolling Your Own DSL in Scala - DZone Java" (2017). The technique consists of creating functions for each command in the DSL. For example, to create the `in(state)` command, an `in` function was created.

To chain commands, a new class containing a new command is returned from the original command's function. For example, to implement the `when(method) goto ``nextstate"` command, the `when` method is defined, which returns a `Goto` class which has a `goto` function. The simplified code for this command is shown in the following listing:

```
def when(methodSignature: String) = {
    new Goto(methodSignature)
  }

class Goto(methodSignature: String){
    def goto(nextState: String) ={
      new At()
    }
  }
```

Notice that at the end of the `goto` method a new `At` class is returned to enable the conditional method transition syntax using the `at` keyword.

5.1.2 Processing data

After the user has written their protocol description, they will run it. This produces a state machine containing the information about the protocol. This is then serialized and written to a file for the checker to read.
In this section, the classes used to construct the state machine are described and the choices of data structures justified. Then the code used to create the state machine is explained.

Classes

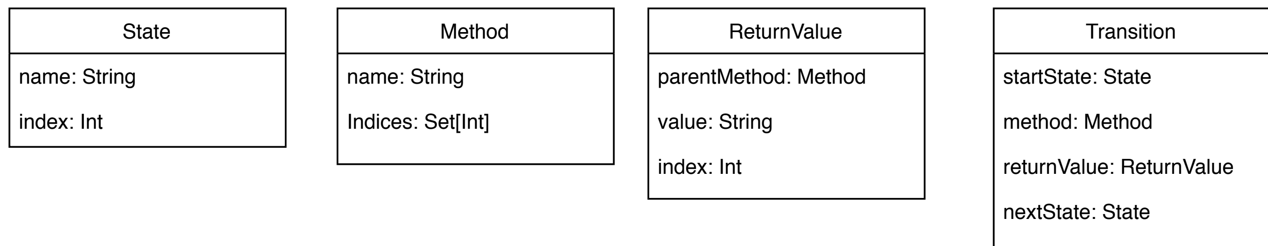


Figure 4: The classes involved in the implementation of the protocol language.

Figure 4 shows the classes used to implement the protocol language. These are explained in detail below.
The State class contains the name of the state and its index into the state machine.
The Method class contains its name, and a set of indices into the state machine. This is a set because a method might have multiple return values which each have their own index into the state machine.
The ReturnValue class is created when the protocol defines a transition which is dependent on a method returning a specific return value. A ReturnValue is linked to the method via the parentMethod attribute. It has a value and

an index into the state machine. Note that the indices of a Method are constituted of the indices of the return values linked to it.

A Transition describes a change from one state to the next in the protocol. It corresponds to the arrows in the diagram of the running example (figure 3). A Transition is constituted of the state it starts at, the (method, return value) pair which enables the transition and the state it ends up in (nextState). Transitions are used to construct the state machine.

Data structures

The main data structure created by the protocol language is the state machine. This captures all the transitions between the states described in the protocol. A two dimensional array of states represents the state machine. On the y axis are the states, identified by index; and on the the x axis are (method, return value) pairs, identified by the index of the return value. The state machine for the running example is shown in table 1.

	walk()	sleep(Int)	run()	run():true	run():false	slow()	stop()	wake()
init	Started	Sleeping	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*
Started	Walking	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*
Walking	*undef*	*undef*	*undef*	Running	Walking	Slow	*undef*	*undef*
Slow	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	init	*undef*
Sleeping	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	init
Running	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*	*undef*

Table 1: Internal representation of a state machine as a two dimensional array of states. Here, “undef” (undefined) means that a next state is not defined for the specified transition, i.e. that it is an illegal transition. For example there is no available “stop()” method in the “init” state, so the 7th entry of the 1st row is undefined.

Sets are used to hold the states, methods, and return values. Sets only allow unique objects, which is desired since the protocol should not have multiple copies of states, methods or return values. Additionally, sets are efficient and have O(1) complexity for all operations, since they are implemented with hash tables (“Data structures and their approximate time complexity for some common” (2011)).

High level code

When the protocol is run, `in` and `when` methods create `State` and `Method` objects and add them to their respective sets.

Creating `Transition` objects is a little more involved. Since `goto` only takes the method name as an argument, it does not know whether or not an `at` command is used in the transition. This is problematic because a `Transition` object must contain a return value (if an `at` command is used) to be properly defined. So, in the code:

```
1 when("run()") goto
2 "Running" at "true" or
3 "Walking" at "false"
4 when ("walk()") goto "Walking"
```

`goto` cannot know that the method "run" has multiple transitions as it can only see the method name "Running".

To solve this, `goto` creates a Transition object with the default value of `Any` as a return value. "Any" corresponds to a transition where no return value is defined in the protocol, as is the case for the `walk()` function in the code above.

Afterwards, when the `at "true"` method is encountered, the `at` method can add another transition to the run method which will take into account the return value (`"true"`).

This will keep adding transitions until no more `at` methods are encountered.

A similar problem appears at the `or` function which will only know the next state and not the return value yet until the `at` statement is reached. In this case a transition with a temporary default value of "undefined" for the return value is added to the set and the `at` later overwrites it.

To construct the state machine, an array of the size of the state $\times$ returnValue sets is created. The transitions are then iterated over and each puts in the relevant next state for each (state, returnValue) entry, to fill the array.

5.1.3 Testing

Writing tests for the DSL was done with ScalaTest "ScalaTest" (2009). Behavioural tests were done, taking a protocol as input and checking the data structures were created as expected.

5.2 Protocol checker

The protocol checker serves to make sure the defined protocol is followed in the user's code. In this section we will present an overview of its function, classes and data structures; then we will explain the implementation of each of its features.

5.2.1 Overview

The checker tracks the state of all instances of classes and objects with a protocol in the code to ensure the protocols are followed correctly. It works by mirroring the execution of the code and tracking all the instances in a global data structure.

Classes

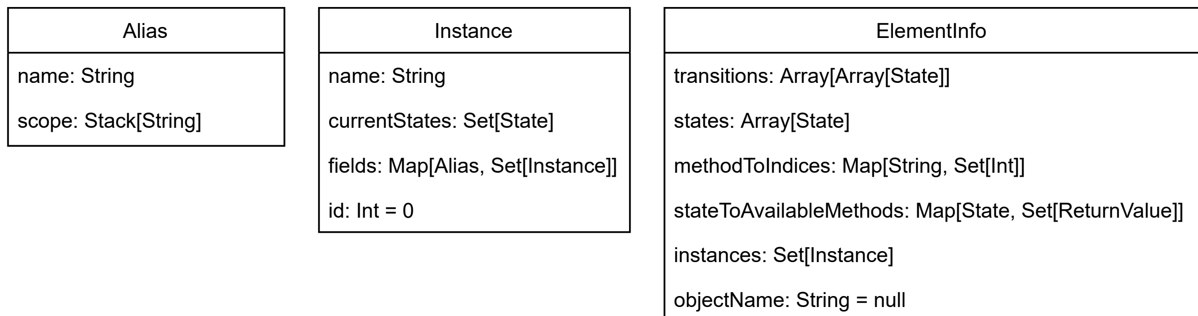


Figure 5: The main classes involved in the implementation of the checker.

The main classes for the checker are shown in figure 5. The three classes are: Instance, Alias and ElementInfo. Instances correspond to objects in memory. They are named "Instance" rather than "Object" because Scala has an `object` qualifier, referring to a singleton object. Using "Object" as an internal name would have caused confusion, so "Instance" was used instead. Instances can contain information about a class or an object. These two will be henceforth referred to as "elements" when there is no need to differentiate between them.

An Instance contains a (name, id) pair which uniquely identifies it. The "name" is simply the name of the element (e.g. Cat) and the id is incremented each time one is created. So, for example, in the code:

```
1  class Cat{...}
2
3  object Main extends App{
4      val cat = new Cat()
5      val kitten = new Cat()
6  }
```

Three instances are created (the '@' symbol here means nothing and is just used for legibility and resemblance to the Java system for naming objects):

- Main@0
- Cat@0
- Cat@1

The “currentStates” attribute is a set of states representing the possible states the Instance can be in at a certain point in the program.

The “fields” represent the fields of the element. These are all of the variables which are defined in a class or object. The fields are kept in a map of their alias (name and scope) to the set of instances they could be pointing at. For example, in the program above (figure 5.2.1), the “Main@0” Instance would have two fields: cat and kitten. So its “fields” would look like this:

```
Main fields:
    cat -> Cat@0
    kitten -> Cat@1
```

An Alias contains a name and a scope which as a pair give it a unique identifier as a field in an Instance. A scope is a stack of Strings identifying where in the program the alias was defined.

Here is an example of a class defining multiple fields with the same name but different scopes:

```
1  class Cat{
2      var friend: Cat = null
3      def decoyFriend(friend:Cat){
4          val friend = new Cat()
5      }
6  }
```

The checker treats all method parameters and instances defined inside a method as fields, so in this case, the Cat instance will have the fields:

- friend with scope “Cat”
- friend with scope “Cat, decoyFriend”
- friend with scope “Cat, decoyFriend, body”, where the “body” serves to differentiate between function parameters and fields defined inside the method body.

Multiple fields might point to the same instance, in which case the instance is “aliased”.

For example, in the following code:

```
class Cat{...}

object Main extends App{
    val cat = new Cat()
    val kitten = cat
}
```

both the “cat” and “kitten” fields point to the same Cat instance. They are called “aliases” of the Cat Instance, and are represented with the `Alias` class.

A visual representation of the code above is shown in figure 6:

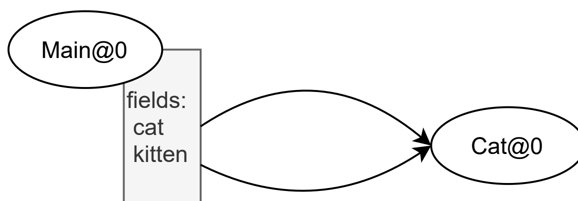


Figure 6: Visual representation of aliasing the Cat@0 instance inside of the Main object. The “cat” and “kitten” fields of the Main@0 instance are aliases of the Cat@0 instance. The nodes represent instances. The rectangle under an instance represents the fields attached to that instance. The arrows show a mapping between the field alias and an instance.

ElementInfo (see figure 5) serves three purposes:

- maintaining a set of instances for each element (inside the “instances” field)
- holding the protocol information of the element (in all the other fields, except “objectName”)
- differentitating between an object and a class, with the “objectName” field

The protocol information is only present in elements which have a protocol attached, otherwise all those fields are null. This protocol information is simply what was described in the protocol. The “objectName” is only defined if the element is an object.

Traversal classes

Function
name: String
params: ArrayBuffer[(String, String)]
body: Trees#Tree
scope: Stack[String]
returnType: Trees#Tree
stateCache: Map[ArrayBuffer[(String, Set[String], Set[State])], ArrayBuffer[Set[State]]]
returned: Option[Set[Instance]]

Element
elementType: String
params: ArrayBuffer[(String, String)]
body: Seq[Trees#Tree]
scope: Stack[String]
isObject: Boolean = false
initialised: Boolean = false
id: Int = 0

Figure 7: Function and Element classes which are necessary for traversal of the user code. Their main purpose is to let the checker jump to their body when needed.

The two classes shown in figure 7 are primarily used for traversing the code. They are quite similar and both contain a name/elementType, params (parameters), body and scope.
The body is stored in the object so it can be jumped to easily when encountering a constructor or function call in the user code. The scope is used to locate the correct function/element given a tie on the name or elementType. For the Function there is a return type; a cache which will be explained further in section 5.2.6; and a “returned” field which can store what the function returns.
The Element contains a boolean indicating whether or not it is an object (as opposed to a class), which works in tandem with “isInitialised” which indicates whether the object is initialised. It also contains an id field which is used to assing a unique id to each instance of that element’s type.

Data structures

The main data structure in the checker is a map, called “trackedElements”, of elementType (String) to ElementInfo. This is a global structure and holds the information about all the different instances in the program. It contains some redundant information: the types are present twice on the outermost level and as names inside the Instances. We decided to have this for faster and easier lookup of a type when encountered.
Here is a diagram to exemplify “trackedElements” as it would be after going through program 5.2.1 (protocol information is removed here for clarity, and the vertical bars represent the ElementInfo objects):

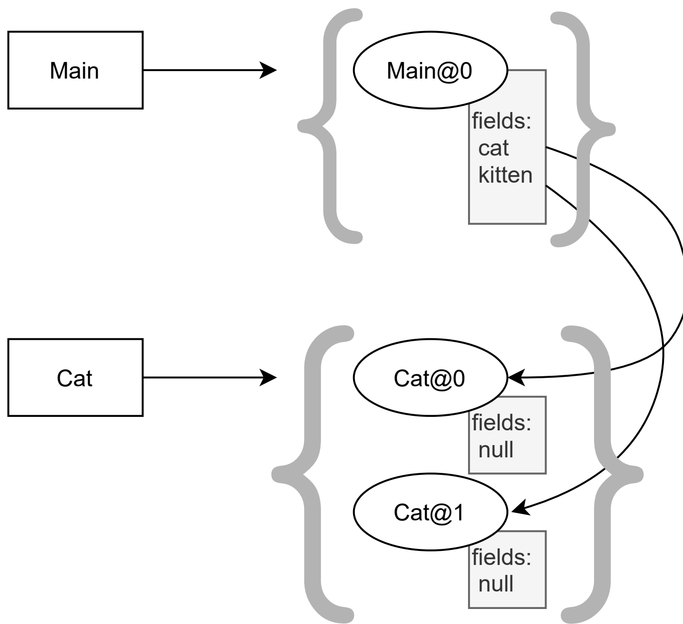


Figure 8: Visual representation of the trackedElements map after going through a program with two cats defined inside a Main object. The white rectangles represent element types which are the main entries into the tracked elements map. They are pointing to the set of instances of that type. The white circles represent instances. They have gray rectangles underneath them containing their fields which point to other instances if they are not null.

Generally, sets are used whenever possible for the same reasons as discussed in section 5.1.2. The states are kept in an array since this is the format they are in when de-serialised and we have enough information to access the elements in O(1) time from the state indices.
In the following sections we will first look at how fields are processed in the checker, and then explain how each of the checker's other features are implemented.

5.2.2 Fields

Any element could contain fields which have a protocol, so all fields are tracked with the checker. This is done through the "fields" attribute of the Instance class. Whenever a variable is created, it is added as a field of the containing element.
To assign a created variable to the correct element, the current containing element is tracked throughout the program. This is only changed when entering the main object or an element constructor.
For the rest of this section, all the created variables are assigned as fields of the Main object. This is not explicitly written, for clarity. Instead, the variables are just treated as unbound Aliases. Additionally, where obvious which instance an alias is pointing at, it will simply be referred to as "alias instance", rather than "the instance which

alias is pointing to".

5.2.3 Basic checking

Finding the entry point of the code

In Scala there are two ways for the user to define an entry point to their code. The first way is to create a main function using `def main(args: Array[String])`. This causes the code entry point to be the object that `main` is defined in. The code will first execute the code of the object (excluding the code in the main function), and then the code in main. For example, the code:

```
1 object MainObject{
2     def main(args: Array[String]): Unit ={
3         println("world")
4     }
5     println("hello")
6 }
```

would first go into `MainObject` and print out "hello", and then go into main and print out "world". The second way a user can create an entry point is by creating an object which extends "App", in which case all the code inside the object will be executed. For example the code:

```
1 object Main extends App{
2     println("hello world")
3 }
```

will print out "hello world".

To find the entry point of the code, the checker goes through the code until it finds an object. If the object extends App it simply goes through the code in the object. If not, it tries to find a main method inside the object body. If a main method exists, it first goes through the enclosing object's code and then through the main method.

Method order checking

When the checker encounters a method call on an alias, it checks the instance(s) pointed to by the alias. For each instance, it gets which state the instance is in and which method is called, then it can index the state machine array to get the next state, or if the transition is invalid: "Undefined". If the transition is valid, it can mutate the state of the instance. Otherwise it throws an error indicating where and why the invalid transition happened. The error message is of the form:

```
"Invalid transition in instance with alias(es) aliasnames of type elementname from state(s) states with method methodname
 in file filename at line linenumber. Possible methods to use in this state are: possiblemethods"
```

In this code, written in file main.scala (using our running example and assuming we have a Cat class defined in another file):

```
1 object Main extends App{
2     val cat = new Cat()
3     cat.sleep(1)
4     cat.sleep(1)
5 }
```

the Cat@0 instance calls the sleep method twice which is not allowed. The error message would be:

```
"Invalid transition in instance with alias(es)TreeSet(cat)of type Cat from state(s)TreeSet(Asleep)with method sleep
()in file Main.scala at line 4. Possible methods to use in this state are: Set(wake())"
```

Objects

In Scala, "object"s can be defined which act like singletons. The checker lets users define protocols for them as well as for classes.

They act much like classes with one difference: objects do not require an initialisation line to have an instance, they exist throughout the program.

Thus, the checker creates the object from the start as an instance. Because there is no explicit initialisation, the constructor code inside the object is called when the object first appears in the code.

This only happens the first time the object appears which is what the "initialised" field of the Element serves to keep track of. This ensures the constructor is only traversed once.

Termination of the protocol

Instances must be in the "end" state at the end of the program for the program to be valid.

The checker goes through all of the instances (with a protocol) at the end of the program and checks their only possible state is the "end" state.

If any instance is not in the "end" state, this error is thrown:

```
"Instance of type instanceType with alias(es) aliases did not necessarily reach its end state. At the
    end of the program it was in state(s) states"
```

For example, in the following code:

```
object Main extends App{
    val cat = new Cat()
    cat.sleep(1)
}
```

the Cat instance with "cat" as an alias did not reach the end state, since it only transitioned from the "init" to the "Asleep" state.

The error message thrown is:

```
"Instance of type Cat with alias(es) TreeSet(cat) did not necessarily reach its end state. At
    the end of the program it was in state(s) Set("Asleep")"
```

5.2.4 Control flow structures

In this section, we will look at the different kinds of loops, if-else statements, try-catch-finally statements, and break statements.

For loops

For loops are structured with a generator and a body:

```
for(generator){
    body
}
```

The generator is of the form (x ← range), where x will vary according to the range.

Just from analysing the code, the checker cannot necessarily know how many times the loop will execute (e.g. if the generator is `(x <- getRandomRange())`), where `getRandomRange()` returns a random range). Therefore, is it assumed that the for loop will run from zero to infinity times.

Therefore, after analysing a loop we cannot know for sure which state the instances will be in. However, it is possible to determine which states the instances **could** be in. These can then all be added to the set of current (possible) states. Subsequently, when a method is called on an instance, the current states can be checked to see if there is a valid transition for all of them with the called method.

To get the possible states, the checker keeps track of the states the instances are in at the end of each loop iteration. It does this by maintaining a map of instances to an arrayList of sets of states. A list of **sets** is maintained

because each instance may have multiple possible states at the end of each loop iteration.

The algorithm is as follows:

1. The checker first adds all the instances and their current states to the map.
2. Then it processes the loop body and adds the states the instances are in at the end of it to the map.
3. It repeats this until the protocol for all instances has looped, which will correspond to all the lists of sets in the map having duplicate sets of states in them.
4. Finally, it adds all the possible states to the current states of the instances.

If the states do not loop, then an error will be raised during the loop, possibly after multiple iterations.

One should also note that aliases defined inside a for loop are scoped inside the loop since loops have their own scopes.

Example

	Code	**Map**	**Instances**
1	`object Main extends App{`		
2	`    val cat = new Cat()`		`Cat@0;init`
3	`    for(x<-1 to 10){`	$Cat@0 \rightarrow init$	`Cat@0;init`
4	`        cat.sleep()`	$Cat@0 \rightarrow init$	`Cat@0;Sleeping`
5	`        cat.wake()`	$Cat@0 \rightarrow init$	`Cat@0;end`
6	`        cat.reset()`	$Cat@0 \rightarrow init$	`Cat@0;init`
7	`    }`	$Cat@0 \rightarrow init, init$	`Cat@0;init`
8	`}`		

After the checker has gone through the code above, it can add the states in the map to the "cat" instance. Here it only adds "init" since the cat is always in that state before and after the loop. Since the "cat" instance already had state "init" in its set of states, this operation is redundant.

The loop analysis ends after the first pass since the protocol has a init-Sleeping-end-init loop. It is apparent that this for loop will cause no problems no matter how many times it is run.
Here is another example with a longer loop (line numbers are removed since multiple passed of the loop are shown and they would be confusing):

Code	Map	Instances
`object Main extends App{`		
`val cat = new Cat()`		`Cat@0;init`
first pass of the loop		
`for(x <- 1 to 10){`	`Cat@0->init`	`Cat@0;init`
`cat.walk()`	`Cat@0->init`	`Cat@0;Started`
`}`	`Cat@0->init,Started`	`Cat@0;Started`
second pass of the loop		
`for(x <- 1 to 10){`	`Cat@0->init,Started`	`Cat@0;init`
`cat.walk()`	`Cat@0->init,Started`	`Cat@0;Walking`
`}`	`Cat@0->init,Walking`	`Cat@0;Walking`
third pass of the loop		
`for(x <- 1 to 10){`	`Cat@0->init,Walking`	`Cat@0;init`
`cat.walk()`	`ERROR`	`ERROR`
`}`		
`}`		

Here an error will be thrown since there is no transition from state Walking with the `walk` method.

For-yield loops

For-yield loops are of the form: `for(generator) yield expression`

In terms of instances, this works the exact same way as `for` loops, replacing "body" with "expression".

While loops

While loops are of the form:

```
1 while(condition){
2     body
3 }
```

They are also processed in the same way as for loops.

Do-while, do-while(true) and while(true) loops

Do-while loops are of the form:

```
1 do{
2     body
3 } while(condition)
```

Do-while(true) and while(true) loops are special cases of do-while and while loops.
The body of these loops is definitely executed once.
Therefore, the checker only initialises the map with the states of the instances after the first pass through the loop. In this way, the state the instances were in before the loop body is not conserved.

If/Else statements

When encountering an if-else statement, two branches of execution are possible:

- through the if body, or
- through the else body.

The checker must put the instances through both possible branches and update their states with both possible outcomes.

Importantly, it must put the instances through the else body as they were before they were put through the if body. To achieve this, the checker copies the instances before going into the if-else statement, then puts both sets of instances through the if-body and the else-body. This gives two new sets containing both possible outcomes. finally, the checker merges both sets together.

Example

	Code	Instances	If-instances	Else-instances
1	`object Main extends App{`			
2	`    val cat = new Cat()`			
3	`    var x = 1`	Cat@0;init		
4	`    if(x == 1)`	Cat@0;init		
5	`        cat.sleep(x)`	Cat@0;init	Cat@0;Asleep	
6	`    else`	Cat@0;init	Cat@0;Asleep	
7	`        cat.walk()`	Cat@0;init	Cat@0;Asleep	Cat@0;Started
8	`}`	Cat@0;Asleep,Started		

The `if` body puts Cat@0 into state "Asleep" and the `else` puts it into state "Started". Then these states are merged in the instances so Cat@0 is now in the two possibles states: "Asleep" and "Started".
Note that the "init" state is not maintained since in an if-else statement at least one branch must be taken.
Notice also that there is no method after this which "cat" could call which would be legal since states Asleep and Started have no method transitions in common.

Lone if statements

The Scala compiler (scalac) converts all lone if statements to if-else statements so no change to the processing is needed.

Try-catch-finally statements

Try-catch-finally statements are dealt with in a basic way. The code goes through the try and then the finally body. This does not account for exceptions, which could miss errors in the protocol if there is code in the catch body, or in the case below:

```
1  val cat = new Cat()
2  try{
3      //other code which might throw a FileIOException
4      ...
5      cat.sleep(1)
6  }
7  catch{
8      case FileIOException => ...
9  }
10 finally{
11     cat.wake()
12 }
```

Here, if no exception is thrown, the Cat@0 will simply transition through the Asleep and init states with methods sleep and wake. However, if an exception is thrown before the `cat.sleep(1)` line, then the `cat.wake` line would be executed while the cat was in the init state which is invalid. This is not caught by the checker in its current state and is discussed in the further work section (see section 8).

Break statements

In Scala, there are no native break or continue statements. Instead, the **breakable** library is used which enables the user to define a breakable block.

The syntax is as follows:

```
1  breakable{
2      codeA
3      break()
4      codeB
5  }
```

The flow of execution is through codeA and then out of the breakable block. codeB is not executed.

The breakable block can also have a label which enables one to use a labelled break statement referring to it. In this code:

```
1  outer.breakable{
2      codeA
3      inner.breakable{
4          codeB
5          outer.break()
6          codeC
7      }
8      codeD
9  }
```

codeA is executed, then codeB, and then `outer.break()` breaks out of the `outer.breakable` block and neither codeC nor codeD are executed.

Here is an example where various control flows are possible:

```
1  var x = 0
2  outer.breakable{
3      do{
4          inner.breakable{
5              if(x == 0){
6                  x += 1
7                  codeA
8                  inner.break()
9              }
10             else{
11                 codeB
12                 outer.break()
13             }
14         }
15     } while(true)
16 }
```

Here the if loop body will keep executing until x is not equal to 0, at which point the else body will execute and break out of the outermost `breakable`.

To process break statements, the checker keeps track of what states the instances were in at each break statement, and recalls that information at the end of the `breakable` block. This covers the possible execution flows in the code.

At a break statement, the checker also returns an empty set of instances to the enveloping block of code. For example, in the code above, an empty set would be returned at the end of the `if` code block, reflecting the fact that breaks kill code beneath them in a code block.

At the end of a `breakable` statement, the checker merges the instances as they were at each break statement, and as they are at the end of the breakable block.

A map of (breakable label $->$ array of instances) is created, which contains the saved instances for each `label.break()` statement. After dealing with a breakable block, its entry in the map is removed so that further usage of the same label goes unhindered.

Note that since the Scala compiler changes unlabelled `breakable` statements to `Breaks.breakable`, all breakable statements are actually labelled and can be treated the same way.

Here is an example demonstrating how the checker works with break statements (here the underlying Cat@0 instance is forgone for clarity, it would be what the "cat" alias is pointing to):

	Code	Instances	Breaks map
1	`var x = 0`		
2	`val cat = new Cat()`	`cat;init`	`empty`
3	`outer.breakable{`	`cat;init`	`outer->null`
4	`  do{`	`cat;init`	`outer->null`
5	`    inner.breakable{`	`cat;init`	`outer->null; inner->null`
6	`      if(x == 0){`	`cat;init`	`outer->null; inner->null`
7	`        x += 1`	`cat;init`	`outer->null; inner->null`
8	`        cat.walk()`	`cat;Started`	`outer->null; inner->null`
9	`        inner.break()`	`empty`	`outer->null; inner->cat;Started`
10	`      }`	`empty`	`outer->null; inner->cat;Started`
11	`      else{`	`cat;init`	`outer->null; inner->cat;Started`
12	`        cat.sleep(1)`	`cat;Asleep`	`outer->null; inner->cat;Started`
13	`        outer.break()`	`empty`	`outer->cat;Asleep; inner->cat;Started`
14	`      }`	`empty`	`outer->cat;Asleep; inner->cat;Started`
15	`    }`	`cat;Started`	`outer->cat;Asleep`
16	`  } while(true)`	`cat;Started`	`outer->cat;Asleep`
17	`}`	`cat;Started,Asleep`	`empty`

In the code above, we see that on line 9, the instances reset to empty upon reaching the "break()" statement. This lasts till the end of the `if` block, after which the instances are reset to the state they were in before the `if` block, to be passed into the `else` block. On the same line, the breaks map is updated with the state of the instances. This happens on line 13 as well.

Subsequently, on line 15, the "inner" entry of the breaks map is merged with the empty instances set. This happens again on line 17 with the "outer" label.

5.2.5 Aliasing and assignments

When writing code, it is possible to call the same object in memory by different variable names. e.g.

```
1 val cat = new Cat()
2 cat.sleep()
3 val kitten = cat
```

where both "cat" and "kitten" point to the same object in memory with state "Asleep"

To represent this in the checker, multiple Alias objects can point to the same Instance object.

In the following code, both the "cat" and "kitten" aliases point to the Cat@0 instance and can therefore both influence its current state.

The | character is used to separate multiple aliases which are pointing to the same Instance(s).

For example,

```
cat | cat2 -> Cat@0;init
```

means that "cat" and "cat2" are aliases of instance Cat@0, which is in state "init".

Code	**Aliases pointing to instances**
1 `val cat = new Cat()`	`cat -> Cat@0;init`
2 `cat.sleep()`	`cat -> Cat@0;Asleep`
3 `val kitten = cat`	`cat	kitten -> Cat@0;Asleep`

Assignments

Aliases may point to different instances according to assignment statements.

These come in four types:

- A: the assignment of a new variable to a new object (val/var x = new Y())
- B: the assignment of a new variable to an existing object (val/var x = y)
- C: the assignment of an existing variable to a new object (x = new Y())
- D: the assignment of an existing variable to an existing object (x = y)

In case A, the checker creates a new instance with alias “x” pointing to it.

In case B, the checker adds a new alias “x” and makes it point to the existing instance which is being pointed to by “y”.

In case C, the checker stops alias “x” from pointing to its current instance (since it is being overwritten), and creates a new instance for it to point at.

In case D,the checker stops alias “x” from pointing to its instance, and makes it point to the existing instance which is being pointed to by “y”.

Full examples for each of these cases can be found in the Appendix, section 8.4.

Aliasing and if-else statements

Aliases point to a set of instances as opposed to a single instance, because a variable might be pointing to multiple instances.

This can happen with aliasing in an if-else statement as shown below (the states of the instances are dropped here for clarity, they would all be “init”):

	Code	Instances	If/Else-instances
1	`object Main extends App`		
2	`{`		
3	`  val cat1 = new Cat()`	`cat1->Cat@1`	
4	`  val cat2 = new Cat()`	`cat1->Cat@1 cat2->Cat@2`	
5	`  var x = 1`	`cat1->Cat@1 cat2->Cat@2`	
6	`  if(x == 1)`	`cat1->Cat@1 cat2->Cat@2`	`cat|cat1->Cat@1 cat2->Cat@2`
7	`    val cat = cat1`	`cat1->Cat@1 cat2->Cat@2`	
8	`  else`	`cat1->Cat@1 cat2->Cat@2`	`cat1->Cat@1 cat|cat2->Cat@2`
9	`    val cat = cat2`	`cat1->Cat@1 cat2->Cat@2`	
10	`}`	`cat|cat1->Cat@1 cat|cat2->Cat@2`	

We see that on line 10, the “cat” alias now points at both the Cat@1 and Cat@2 instance and method calls on that alias will cause both instances to transition state.

5.2.6 Functions

To process function calls, a scope system is needed so that aliases with the same name do not all change state when only one has a method called on it.

For example, lets look at what we want to happen in the code:

```
object Main extends App{
    val cat = new Cat()
    bothSleep(cat)
    cat.sleep()

    def bothSleep(kitty:Cat){
        val cat = new Cat()
        cat.sleep()
        kitty.sleep()
    }
}
```

There are two problems to handle in the above code:

- the two "cat" variables are distinct in the code and we need a system to differentiate between them: the scope system.
- when "kitty" is used on line 9 we want this to change the state of the Cat@0 instance created on line 2.

To handle the first problem, the current scope is kept track of while analysing the program. The current scope is composed of a stack of strings which is pushed to when entering an object, class, loop or method, and popped when leaving. When a variable is initialised, the current scope is given to it.
In the code above, two cat aliases are created with scopes:

- *main*; and
- *main, bothSleep*

Then, when the code encounters the"cat.sleep()" line, it will find the alias with the name "cat" which has the closest scope to the current scope, and mutate the state of the instance(s) it points to.

For the second problem, we can add a set of assignments at the beginning of a function. Each assignment corresponds to creating a new field pointing to a parameter passed into the function.
For example, in the code above, the `val kitty = cat` line is added when the function `bothSleep` is called.
This makes "kitty" point to Cat@0. Then, when `kitty.sleep()` happens, Cat@0's states can be changed. Aliases are deleted once they go out of scope.
Looking at the instances set alongside the code and expanding the function call, we have (the ":" character prefaces the scope of an alias, the numbers are removed since we are going into the method call):

Code	**Instances**
`val cat=new Cat()`	`cat:main -> Cat@0;init`
`bothSleep(cat)`	
`Inside the method call`	*`Inside the method call`*
`>val kitty=cat`	`cat:main|kitty:main,bothSleep->Cat@0;init`
`>val cat=new Cat()`	`cat:main|kitty:main,bothSleep->Cat@0;init cat:main,bothSleep->Cat@1;init`
`>cat.sleep()`	`cat:main|kitty:main,bothSleep->Cat@0;init cat:main,bothSleep->Cat@1;Sleeping`
`>kitty.sleep()`	`cat:main|kitty:main,bothSleep->Cat@0;Sleeping cat:main,bothSleep->Cat@1;Sleeping`
`After the method call`	`cat:main->Cat@0;Sleeping` *`kitty and cat defined in the method are deleted`*
`cat.sleep()`	`ERROR since we cannot call sleep twice on the same instance`

Clashing method body and protocol transition

There is an interesting case if an instance calls a method in its protocol which also mutates the state of the instance in its body. For example in the code:

```
1     object Main extends App{
2         val cat = new Cat()
3         cat.walk()
4
5         class Cat{
6             def walk(){
7                 sleep(1)
8             }
9             def sleep(nb:Int){
10                println(s"Sleeping $nb hours")
11            }
12        }
13    }
```

`cat.walk()` is calling the “sleep” method. This leads to a problem because according to the protocol, the cat instance is transitioning from state “init” to state “Started” through the `walk()` function, but the body of the walk function is putting the cat instance from the “init” to the “Asleep” state.
If nothing was done to cater to this situation, the checker would go through the function, change the state of “cat” to “Asleep” and then try to transition from that state using `walk()` This would lead to an error, which is incorrect according to the protocol, since a Cat in state “init” should be able to call the walk method.

One possible solution to this would be to force the cat instance into the Started state, assuming that the protocol is correct, and ignoring the state mutation on cat from the body of the walk method.

However, this would be leaving a glaring inconsistency in the defined protocol and the method body which the user has defined.
Since this is probably a mistake on the user's part, an error should be thrown so that the user knows to edit their method body.

To check for this error, the checker gets the state of the instance after:

- transitioning through the method according to the protocol
- transitioning through the method according to the method body

First of all, it checks whether the state has changed in the method body. If not, it can skip the rest of this error-checking. This is done because if the method body does not mutate the state then there is no issue with following the transition via the protocol after that.

If the method has changed the state, it compares both states after the transitions. If they are the same, then the method's body changes the state in the same way as the method would, and the code is accepted. If not, an error is thrown. The error message is of the form:

`"In file` *filename*`, at line` *linenumber*`, method` *methodname* `did not mutate state of` *aliasname* `as described in the protocol. Expected states` *expectedstates*`, got states` *actualstates*`"`

The error message thrown by the code above is:

```
"In file Main.scala, at line 3, method walk() did not mutate state of cat as described in the protocol. Expected states
 Set(Started), got states Set(Asleep)"
```

Caching and recursion

Functions are cached so that if the same function is called with parameters in the same states as a previous call, it will give the cached result instead of processing the function again.
It does this by attaching a map to the function which goes from a list of (parameter name, states) pairs to a list of states.

For example (here the $\rightarrow$ symbol corresponds to a mapping, the numbers are removed because we are going into the method call):

Code	**Cache for makeWalk()**
`object Main extends App{`	
`    val cat = new Cat()`	
`    makeWalk(cat)`	
	`kitty, init ->` *null*
`    ` *inside call*	`kitty, init ->` *null*
`    > kitty.walk()`	`kitty, init ->` *null*
`    > kitty.walk()`	`kitty, init ->` *null*
`    > kitty.slow()`	`kitty, init ->` *null*
`    > kitty.stop()`	`kitty, init ->` *null*
	`kitty, init -> init`
`    makeWalk(cat)`	`kitty, init -> init`
call can be skipped since result is cached	
`    def makeWalk(kitty:Cat)={`	
`        kitty.walk()`	
`        kitty.walk()`	
`        kitty.slow()`	
`        kitty.stop()`	
`    }`	
`}`	

This serves us well for recursion since the cache will be in a $x \rightarrow null$ state if the function has not returned. If it is called again with the same parameters, we know the function is recursive and will no longer change the state of the parameters, so we can skip the call. After that, the function should end normally and the checker code can return from the function and resume, instead of looping forever.

For example:

Code	**Cache for recursive()**
`object Main extends App{`	
`    val cat = new Cat()`	
`    recursive(cat)`	
	`kitty, init -> null`
`     ` *inside call*	`kitty, init -> null`
`    > recursive(kitty)`	`kitty, init -> null`
skip call because cache entry points to null	`kitty, init -> null`
	`kitty, init -> init`
`    def recursive(kitty:Cat)={`	
`        recursive(kitty)`	
`    }`	
`}`	

If the recursive function changes the state of a parameter, then it might recurse several times before the parameter goes back to having the original state, at which point the checker will exit. This works similarly to loops.

5.2.7 Return values

Returning values from functions

In Scala functions, the object returned will be either the one after the `return` keyword or simply the last object present on the last line in the function control flow. For example, in

```
1 def bothSleep(kitty:Cat):Cat{
2     val cat = new Cat()
3     cat.sleep()
4     kitty.sleep()
5     kitty
6 }
```

the function bothSleep() returns the instance "kitty" of class Cat. This code is equivalent:

```
1 def bothSleep(kitty:Cat):Cat{
2     val cat = new Cat()
3     cat.sleep()
4     kitty.sleep()
5     return kitty
6 }
```

To find the return value of a function the checker goes through the function body until it gets to the end while keeping track of the names it encounters.

For example:

Code	Return value
`1 def bothSleep(kitty:Cat):Cat{`	
`2     val cat = new Cat()`	`cat`
`3     cat.sleep()`	`cat`
`4     kitty.sleep()`	`kitty`
`5     kitty`	`kitty`
`6 }`	

The raw return value (e.g. "kitty") is matched against alias names and the actual instance(s) associated with it is captured. This can then be assigned if needed using the code dealing with assignments explained in section 5.2.5, or passed along to be returned by another function. Once this is done, the instances defined inside the function can be deleted.

For example (alias scopes removed for clarity):

Code	Instances
1 `object Main extends App{`	
2 `    val cat = new Cat()`	`cat -> Cat@0;init`
3 `    val cat2 = makeSleep(cat)`	
4 `     inside call`	
5 `    > val kitty = cat (added line)`	`cat|kitty -> Cat@0;init`
6 `    > kitty.sleep()`	`cat|kitty -> Cat@0;Asleep`
7 `    > kitty`	
8 `     leave call, add line below`	
9 `    cat2 = kitty (added line)`	`cat|kitty|cat2->Cat@0;Asleep`
10	`Delete kitty after returning from method`
11	`cat|cat2 -> Cat@0;Asleep`
12 `    def makeSleep(kitty:Cat):Cat{`	
13 `        kitty.sleep()`	
14 `        kitty`	
15 `    }`	
16 `}`	

The code above is taking "cat", making it sleep, and then creating "cat2" and making it point to the first cat.

Return values from If-else statements

In Scala, if-else statements return a value and therefore can be used in assignment statements. For example, the code:

```
1  val x = 0
2  val cat = if(x == 0) cat1 else cat2
```

would assign cat1 to "cat". Here, since the checker cannot know which branch of execution will execute, and therefore what will be returned, it returns both results in an Array and then assigns both of them. In the code above, the checker instances evolve as follows (alias scope is omitted for clarity):

Code	Instances
1 `val cat1, cat2 = new Cat()`	`cat1->Cat@0;init cat2->Cat@1;init`
2 `cat1.walk()`	`cat1->Cat@0;Started cat2->Cat@1;init`
3 `val x = 0`	`cat1->Cat@0;Started cat2->Cat@1;init`
4 `val cat = if(x==0) cat1 else cat2`	`cat1->Cat@0;Started cat2->Cat@1;init`
5 ` if case`	` if case`
6 `> val cat = cat1`	`cat1|cat->Cat@0;Started cat2->Cat@1;init`
7 ` else case`	` else case`
8 `> val cat = cat2`	`cat1->Cat@0;Started cat2|cat->Cat@1;init`
9 ` after if-else`	` after if-else`
	`cat1|cat->Cat@0;Started cat2|cat->Cat@1;init`

The if-else statement has the effect of "cat" pointing to the instances "cat1" and "cat2" were pointing at. Now,

any method call on cat will update both Cat@0 and Cat@1.

Return values in the protocol

In the protocol, users can define a transition which happens when a method returns a certain value (e.g. "run():true"). At compile time, we cannot know which value will be returned from a function. Thus the user has two options:

- 1: they can create a match statement and define different code paths for each possible return value
- 2: they can write the method without a match statement. In that case, the code following the match must be valid given any return value by the method.

Case 1: match statement

Match statements have the following structure:

```
1 expr match{
2     case possible_expr1 => case_body1
3     case possible_expr2 => case_body2
4     ...
5     case possible_exprX => case_bodyX
6     case _ => default_body
7 }
```

for X case statements. The execution path through the match statement is through the `expr`, then through whichever `case` matches `expr`. `case _ =>` is the default path and the `default_body` will execute if none of the other cases match.

If `expr` **is not** a method defined in the protocol, the checker checks `expr`. It then goes through all the case statements using the results from `expr`, merging the result from all case statements together at the end.

If `expr` **is** a method in a protocol, it is not processed immediately, but at the top of each case statement.

For each case statement, the `possible_expr` (the return value of the method) is added to the method so that it can match the (method name, return value) pair defined in the protocol.

For example, in:

	Code	Instances
1	`object main extends App{`	
2	`    val cat = new Cat()`	`cat -> Cat@0;init`
3	`    cat.walk()`	`cat -> Cat@0;Started`
4	`    cat.walk()`	`cat -> Cat@0;Walking`
5	`    cat.run() match{`	`cat -> Cat@0;Walking`
6	`        case true =>`	`cat -> Cat@0;Running`
7	`        case false =>`	`cat -> Cat@0;Walking`
8	`            cat.slow()`	`cat -> Cat@0;Slow`
9	`    }`	`cat -> Cat@0;Walking, Slow`
10	`}`	

the checker will look for the "run():true" (method, return value) pair on line 6, and the "run():false" method on line 7. This means that the cat instance can be in solely the "Running" or "Walking" state inside the case statement which enables a legal transition to the "Slow" state inside the `case false` body.

Case 2: no match statement

When a method is used outwith a match statement and has multiple transitions dependant on return values, all possible states are added after the call.

In the example protocol, the Walking state can lead either back to itself at run():false or to the Running state at run():true. After the "run()" call, the checker would add both the "Running" and "Walking" states to the current states of the instance.

Example

In this example, the underlying Cat@0 instance is left out for clarity, it would be what "cat" is pointing to for the entire program

	Code	Instances
1	`object Main extends App{`	
2	`    val cat = new Cat()`	`cat;init`
3	`    cat.walk()`	`cat;Started`
4	`    cat.walk()`	`cat;Walking`
5	`    cat.run()`	`cat;Walking,Running`
6	`}`	

In the code above, on line 5, both the "Walking" and "Running" states are added to the instances as the return value of the "run()" method is unknown at compile time.

5.2.8 Testing

To test the checker, it is added as an additional phase in the compiler used during testing.
Different pieces of code are then ran through it and automated tests check whether or not an error is thrown, as well as what the error message contains.
All of the features above are tested in this way.

6 Case study

In this section, we will present two realistic applications of Scala-Mungo. Firstly, we will exemplify the aliasing system with an example about processing money. Then, we will show Scala-Mungo's robustness with an implementation of the simple mail transfer protocol (SMTP).

Aliasing example

Let us imagine that a user wants to model the flow of money in their company.

The money should be an object which can have an amount of money added to it, then interest added, and only then should be accessed.

The user decides to model these constraints as the following protocol:

```
1  object MoneyStashProtocol extends ProtocolLang with App {
2    in("init")
3    when("fill(Float)") goto "filled"
4    in("filled")
5    when("applyInterest(Float)") goto "appliedInterest"
6    in("appliedInterest")
7    when("get()") goto "end"
8    in("end")
9    end()
10 }
```

Which is equivalent to the graph on figure 9.

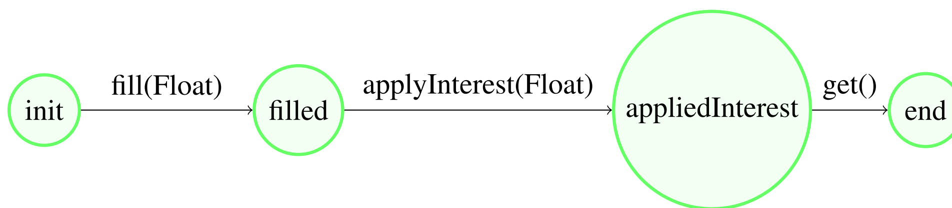


Figure 9: Graph of the MoneyStash protocol.

Then the user wants it to be possible for a manager and database to interact with this money object. The manager to add a salary to the amount, and the database to store the amount of money.

The user would create their classes as follows.

```
@Typestate("MoneyStashProtocol")
class MoneyStash() {
  var amountOfMoney : Float = 0
  def fill(amount : Float ) : Unit = {amountOfMoney = amount}
  def get() : Float = amountOfMoney
  def applyInterest(interest_rate : Float) : Unit = {
    amountOfMoney = amountOfMoney * interest_rate;
  }
}

class DataStorage() {
  var money : MoneyStash = null;
  def setMoney(m : MoneyStash) : Unit = {money = m}
  def store() : Unit = {
    var amount = money.get()
    println(amount)
    // write to the database
  }
}

class SalaryManager() {
  var money : MoneyStash = null;
  def setMoney(m : MoneyStash) : Unit = {
    money = m
  }
  def addSalary(amount: Float) : Unit = {
    money.fill(amount)
    money.applyInterest(1.02f)
  }
}
```

The MoneyStash class is a container for the money and is tied to the above protocol with the `@Typestate` annotation. It must implement all the methods mentioned in the protocol, and can implement others.
The DataStorage class represents an interface to the database and has a field to contain a MoneyStash object.
The SalaryManager class represents the manager and also has such a field.

The user can then write their program, in which the MoneyStash object is shared between the manager and the database. To accomplish this sharing, one MoneyStash object is created and aliased, and given as a field for both the manager and the database (storage).

```scala
object Demonstration extends App {
  val salary = new MoneyStash
  val manager = new SalaryManager
  val storage = new DataStorage

  manager.setMoney(salary)
  storage.setMoney(salary)

  manager.addSalary(5000)
  storage.store()
}
```

In the program, the MoneyStash object (salary) is assigned to the manager and storage on lines 6 and 7. Then the manager uses the `addSalary` method to fill its MoneyStash field and apply interest to it. Finally, the storage saves its MoneyStash through the `store` function.

The global aliasing approach enables the MoneyStash object to be tracked as a single instance, which contains the current state in the protocol. The salary variable and both fields of the manager and storage can then all point to it and make changes to the state.
This leads to unrestricted aliasing, where there are no need for bounds on how much the user can alias a protocolled object, and all aliases can change the state of their instance.
This approach also means that aliasing is transparent to the different classes. Neither MoneyStash, SalaryManager or DataStorage need to have any knowledge that the MoneyStash object is aliased. This lets the user rely on the checker rather than having to add code to specify aliased variables.

Example errors caught

If the user forgets to write line 28 in the classes, an error will be thrown, reflecting the usage of the "get" method before the "applyInterest" method:

```
"Invalid transition in instance with alias(es) Set(money, salary) of type MoneyStash from state
    (s) Set(intermediate) with method get() in file classes.scala at line 15. Possible methods
    to use in this state are: applyInterest(Float)"
```

If the user inverts lines 9 and 10, this will cause the "get" method to be called before the "fill" method. This might not be obvious from inspecting the program since the MoneyStash object is not referred to. However the checker can catch it and throws the following error:

```
"Invalid transition in instance with alias(es) TreeSet(money, salary) of type MoneyStash from
    state(s) TreeSet(init) with method get() in file classes.scala at line 15. Possible methods
     to use in this state are: fill(Float)"
```

SMTP

This example presents an implementation of the simple mail transfer protocol, which requires a large amount of code (735 lines) and complex control flow through while loops, match statements and break statements. The code can communicate with a live email server through TCP sockets. This provides a proof of Scala-Mungo's robustness and use in practical applications.

The simple mail transfer protocol (Postel (1982)), is the protocol used for communication between a client and server to manage and send emails. The example given here is based on the protocol defined in RFC 5321. Communication is driven by text messages exchanged between the client and server.

To start communicating with the server, the client first sends the *EHLO* command, to which the server responds with a success, failure, or error response. The responses from the server are in the form of http codes, such as *250 OK*.

The communication then continues until the client sends the *QUIT* command, which ends communication.

In this example, we will present some of the code for the client side of the SMTP interaction. The full code for this example for be found in the Scala-Mungo github repository (Aliceravier (2021)).

An excerpt from the protocol is shown below:

```
1   ...
2   in("State29")
3   when("send_DATALINEToS()") goto "State30"
4   when("send_SUBJECTToS()") goto "State31"
5   when("send_ATADToS()") goto "State32"
6   in("State30")
7   when("send_datalineStringToS(String)") goto "State29"
8   in("State31")
9   when("send_subjectStringToS(String)") goto "State29"
10  in("State32")
11  when("send_atadStringToS(String)") goto "State33"
12  in("State33")
13  when("receive_250StringFromS()") goto "State18"
14  ...
```

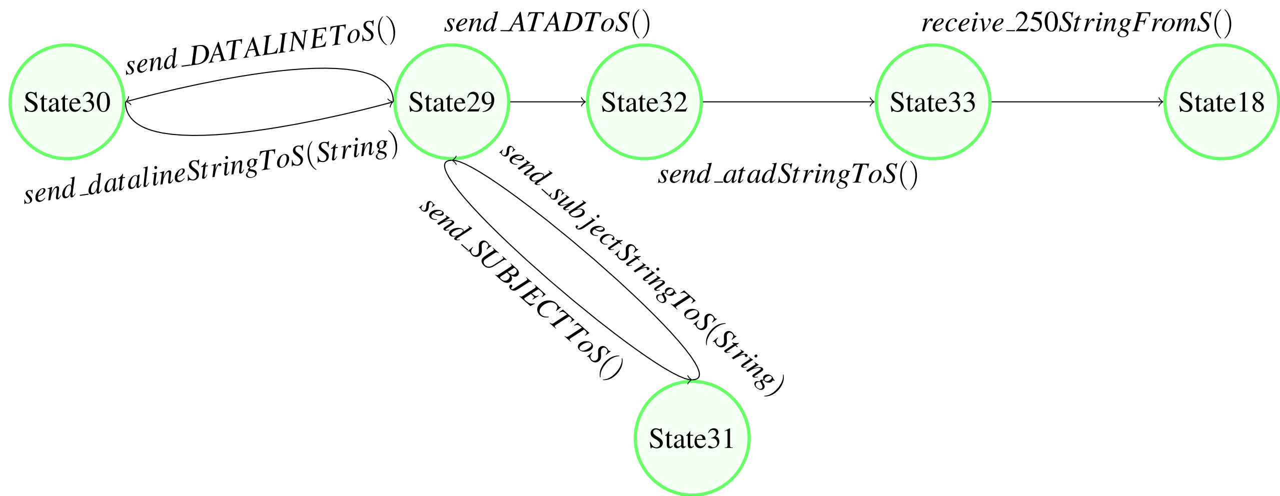


Figure 10: Graph of an excerpt of the SMTP protocol. The nodes represent the states, and the arrows transitions between them. The method names corresponding the the transitions are next to the arrows.

On figure 10 we see a graph of the excerpt from the SMTP protocol. The excerpt describes the part of the protocol which enables the client to create their email.
The client can write a *dataline* to their email, tell the server which *subject* they are addressing it to, or terminate their email using the *atad* command.

The code below shows the piece of the SMTP program using the excerpt of the protocol:

```
1  Z1Inner.breakable{
2      ...
3      do {
4          System.out.print("Choose a label among DATALINE, SUBJECT or ATAD: ")
5          safeRead(readerC) match {
6            case "DATALINE" =>
7              currentC.send_DATALINEToS()
8              System.out.print("Send to S text for DATALINE: ")
9              val payload19: String = safeRead(readerC)
10             currentC.send_datalineStringToS(payload19 + CRLF)
11           case "SUBJECT" =>
12             currentC.send_SUBJECTToS()
13             System.out.print("Send to S text for SUBJECT: ")
14             val payload20: String = safeRead(readerC)
15             currentC.send_subjectStringToS((new SMTPMessage("SUBJECT:" + payload20, CRLF)).
                   toString)
16           case "ATAD" =>
17             currentC.send_ATADToS()
18             currentC.send_atadStringToS("." + CRLF)
19             val payload22: String = currentC.receive_250StringFromS()
20             System.out.println("Received from S: " + payload22)
21             Z1Inner.break()
22         }
23       }
24     } while (true)
25 }
```

The commands to edit the email are written in a do-while(true) loop, so that the user can edit the contents and subject of their email as much as they want. The only way to break out of this loop is to use the *ATAD* command which jumps to the "ATAD" case (line 16) in the code and breaks out of the loop on line 21.

The do-while(true) loop corresponds to the loops seen in figure 10 around States 29, 30 and 31. The break statement reflects exiting the loop to attain state 32.

The SMTP code runs without errors with Scala-Mungo, indicating that all paths the user can take through it are valid and that the protocol will always reach completion.

If line 18 is omitted, the program no longer lets the user terminate their email. This error is caught by Scala-Mungo and the following error message is produced:

```
"Invalid transition in instance with alias(es) TreeSet(currentC) of type CRole from state(s)
    TreeSet(State 32) with method receive_250StringFromS() in file SMTP.scala at line 19.
    Possible methods to use in this state are: send_atadStringToS(String)"
```

This lets the programmer know they have forgotten to write the `send_atadStringToS(String)` method into their

program.

Conclusions

These two case studies show the effectiveness of Scala-Mungo's aliasing approach, as well as the robustness and practicality of the tool. In the aliasing example, Scala-Mungo effectively recognised an error on an object shared between two different classes. In the SMTP example, Scala-Mungo enabled us to create a lengthy, real world program which can communicate with an email server.

7 User testing

In this section, we present the method used to test the Scala-Mungo tool on users.

7.0.1 Aims

We selected four expert users to test the Scala-Mungo tool: people who had experience with protocol checking software. They had all used Mungo and other tools before and could give us feedback on how this tool compared to the state of the art. The decision to test on very little people was made to keep the project in scope, while still getting some feedback on the tool.

The user testing proposed to evaluate the following aspects of the tool:

- Usability
- Intuitiveness
- Correctness

The users were instructed to read the tutorial online ("Scala-Mungo tutorial" (2021), see Appendix section 8.5.1 for the full contents) and then complete the exercises on the same website.

The tutorial consists of setup instructions, a specification for the protocol language, and a detailed usage example.

The exercises and their aims were as follows:

- 1: **Complete the tutorial example.** This was meant to give the user an understanding of the tool and test their setup.
- 2: **Write a protocol from a graph and test it using the provided code.** This was an easy exercise that required some mastery of the protocol language. It was meant to test their understanding of the protocol language.
- 3: **Write a protocol and program, given a written description of a protocol and a class.** This was meant to emulate a real world use case of the Scala-Mungo tool and thoroughly test the user's understanding.

7.0.2 Methods

The users were sent a link to the website and the following list of questions:

- How was the set-up step?

- How difficult were the exercises for you, did the error messages help?
- Approximately how much time did the exercises take?
- Did you find any bugs, what were they?
- What are things which could be improved?
- Any other comments?

7.0.3 Results and discussion

A full list of answers is provided in the appendix, section 8.5.3. Below are the general results.

The setup step was confusing for everyone, with people specifically mentioning:

- The necessary Scala version was missing.
- Having a step-by-step installation process would be useful.

The exercises took users between 15 and 30 minutes, which was fast and shows their understanding of the tool. They generally found the exercises easy and instructive.

Users generally liked the error messages given by the tool (they are described in section 5), notably that it gave the line number where the error occurred. Some parts were confusing however:

- TreeSet() was reported as confusing multiple times. Deleting ”TreeSet” was proposed as a solution.
- The stack trace was unhelpful and should be removed.
- The error showing that a protocol is not terminated confused a user since it shows ”end” in the list of states.
- The recovery functionality of the protocol language which lets methods without parenthesis be recognised led to a confusing error.

The only non-trivial bug discovered was thaat calling a method after the protocol had terminated in the second exercise did not throw an error. This should be investigated.

The most desired change was to remove the need to comment out the `@Typestate` annotation when writing or fixing a protocol. This is necessary at the moment because the ”run” command in Scala runs all of the files in a project. Therefore an incorrect or missing protocol will make the checker throw an error at compile time if the `@Typestate` annotation is present; and the protocol will not be run.

Overall, the users thought the plugin was user-friendly and useful once it was setup.

7.1 Conclusions of the user testing

The results show us that the testers mostly liked the tool and found it useful. Our evaluation shows the following:

- The small time it took users to complete the exercises shows that the tool was intuitive to expert users.
- Correctness was found to be good with only one bug discovered during user testing.
- Usability was lacking due to the difficulties with the setup.

Due to the small amount of testers, these results cannot be confirmed. However, this small test shows promising results. This encourages us to promote larger scale user testing for future work, after the changes to the tutorial are made to make the setup easier.

The changes proposed as a result of the user testing feedback are discussed in the future work section (section 8).

8 Conclusion and future work

8.1 Conclusion

We have presented Scala-Mungo as a tool which incorporates typestates into Scala. It resembles the existing Mungo tool for Java in usage and lets the user define their typestate separately from their code. It differs from Mungo by letting the user write their typestate in Scala rather than JSON, and, importantly, by enabling unrestricted aliasing. This is a novel approach to aliasing and lets the user alias instances at will without requiring them to have any knowledge of aliasing with typestates.

The user testing results revealed that the tool met its primary goals, outlined in the requirements section (section 3), and provided a user-friendly and intuitive experience.

However, some useful features are left to be added to Scala-Mungo, and issues coming out of user testing should be addressed. These are discussed below.

8.2 Future work

This section lists the most useful features which could be added to the checker, and the proposed improvements resulting from the user testing.

8.2.1 Features which could be added

Some features were not implemented into Scala-Mungo because of lack of time, and these should be straightforwards to add. These are:

- Adding support for the explicit "return" statement in a function. This entails not processing the function after the return statement and treating the code after the "return" keyword as the instance returned.
- Adding support for returning instances through match statements. This is very similar to what is being done for if-else and functions at the moment.
- Letting if statements also support methods which have branching paths in the protocol, just like match statements. This would involve adapting the match statement code to if statements.
- Adding support for threading.
- Adding support for functional constructs in the language.

A more difficult feature to add would be to process try-catch-finally statements correctly.

Try-catch-finally with exceptions

It is possible to check this during compilation by having a variable x ranging over the length of the try body, and a y variable ranging over the length of the catch body. Then the plugin could check the first x lines of the try body, the first y lines of the catch body and then the finally body. However, this could get quite lengthy if there is a lot of code inside the try-catch. Another solution would be to have Scala-Mungo do runtime checks as well as compile time checks. In this case, it could be detected on which line an exception was thrown and the try-catch-finally block could easily be analysed.

8.2.2 User testing changes

Following the user testing, the following changes to the Scala-Mungo tool are proposed:

- Re-writing the setup in the tutorial to have more steps and the scala version necessary.
- Changing the error messages to be more user-friendly in line with the tester comments.
- Investigating and fixing the bug causing duplicate method calls after termination of the protocol.
- Finding a way around the need to comment out the `@Typestate` annotation, possibly by following the advice given by a tester.

8.2.3 Further evaluation

Since the user testing done for this report was only of small scale, more evaluation of this tool would be beneficial. We propose the following evaluation ideas:

- User test the tool on many more people (circa 100), separated into groups of experienced programmers in any languages, experienced programmers in Scala, and experts in protocol checking.
- Compare Scala-Mungo to Mungo for Java. This could be done by giving different groups of people either:
  - programs to write using Scala-Mungo
  - programs to write using Mungo, or
  - programs to write using neither tool (as a control group)

  and measuring how long it took them to get to a solution.

References

Aliceravier. (2021, January 9). *Aliceravier/scala-mungo* [original-date: 2020-07-08T18:19:40Z]. Retrieved January 9, 2021, from https://github.com/Aliceravier/Scala-Mungo

Bierhoff, K., & Aldrich, J. (2007). Modular typestate checking of aliased objects. *ACM SIGPLAN Notices*, *42*(10), 301–320. https://doi.org/10.1145/1297105.1297050

Clarke, D. G., Potter, J. M., & Noble, J. (1998). Ownership types for flexible alias protection, 48–64.

Coblenz, M., Oei, R., Etzel, T., Koronkevich, P., Baker, M., Bloem, Y., Myers, B. A., Sunshine, J., & Aldrich, J. (2020). Obsidian: Typestate and assets for safer blockchain programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *42*(3), 1–82.

Cruz-Filipe, L., & Montesi, F. (2020). A core model for choreographic programming. *Theoretical Computer Science*, *802*, 38–66. https://doi.org/10.1016/j.tcs.2019.07.005

Cruz-Filipe, L., Montesi, F., & Peressotti, M. (2019). Choreographies in coq.

Dardha, O., Giachino, E., & Sangiorgi, D. (2012). Session types revisited, In *Proceedings of the 14th symposium on principles and practice of declarative programming - PPDP '12*. the 14th symposium, Leuven, Belgium, ACM Press. https://doi.org/10.1145/2370776.2370794

Data structures and their approximate time complexity for some common. (2011, December 23). Retrieved December 19, 2020, from https://web.archive.org/web/20111223104325/http://essays.hexapodia.net/datastructures

DeLine, R., & Fähndrich, M. (2004). Typestates for objects (M. Odersky, Ed.). In M. Odersky (Ed.), *Ecoop 2004 – object-oriented programming*. Berlin, Heidelberg, Springer Berlin Heidelberg.

Fahndrich, M., & DeLine, R. (2002). Adoption and focus: Practical linear types for imperative programming, In *Proceedings of the ACM SIGPLAN 2002 conference on programming language design and implementation - PLDI '02*. the ACM SIGPLAN 2002 Conference, Berlin, Germany, ACM Press. https://doi.org/10.1145/512529.512532

Gay, S. J., Vasconcelos, V. T., Ravara, A., Gesbert, N., & Caldeira, A. Z. (2010). Modular session types for distributed object-oriented programming. *ACM Sigplan Notices*, *45*(1), 299–312.

Gay, S., & Ravara, A. (2017). Behavioural types: From theory to tools. https://doi.org/10.13052/rp-9788793519817

Gay, S., & Vasconcelos, V. T. (2007). Asynchronous functional session types. *Journal article, University of Glasgow*.

GitHub pages [GitHub pages]. (2020). Retrieved January 11, 2021, from https://pages.github.com/

Honda, K., Vasconcelos, V. T., & Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming, In *ESOP'98*, Springer. https://doi.org/10.1007/BFb0053567

Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P.-M., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H. T., & Zavattaro, G. (2016). Foundations of session types and behavioural contracts. *ACM Computing Surveys*, *49*(1), 1–36. https://doi.org/10.1145/2873052

Kouzapas, D., Dardha, O., Perera, R., & Gay, S. J. (2018). Typechecking protocols with mungo and StMungo: A session type toolchain for java. *Science of Computer Programming*, *155*, 52–75. https://doi.org/10.1016/j.scico.2017.10.006

Militão, F., Aldrich, J., & Caires, L. (2010). Aliasing control with view-based typestate, In *Proceedings of the 12th workshop on formal techniques for java-like programs - FTFJP '10*. the 12th Workshop, Maribor, Slovenia, ACM Press. https://doi.org/10.1145/1924520.1924527

Montesi, F. (2014). Choreographic programming.

Montesi, F., & Yoshida, N. (2013). Compositional choreographies (P. R. D'Argenio & H. Melgratti, Eds.) [Series Title: Lecture Notes in Computer Science]. In P. R. D'Argenio & H. Melgratti (Eds.). D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, & G. Weikum (**typeredactors**), *CONCUR 2013 – concurrency theory*. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg, Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-40184-8_30

Needham, R. M., & Schroeder, M. D. (1978). Using encryption for authentication in large networks of computers. *Communications of the ACM*, *21*(12), 993–999. https://doi.org/10.1145/359657.359659

nitnelave. (2019). Protenc.

Ornela Dardha, A. R., Mathias Jakobsen. (2021). *Global typestate analysis for aliased objects* [work in progress]. work in progress.

Parr, T. (2012). *The definitive ANTLR 4 reference* [OCLC: ocn802295434]. Dallas, Texas, The Pragmatic Bookshelf.

Postel, J. (1982). *Simple mail transfer protocol*. Retrieved January 10, 2021, from https://tools.ietf.org/html/rfc821

Rolling your own DSL in scala - DZone java [Dzone.com]. (2017, March 17). Retrieved December 19, 2020, from https://dzone.com/articles/rolling-your-own-dsl-in-scala

Saini, D., Sunshine, J., & Aldrich, J. (2010). A theory of typestate-oriented programming, In *Proceedings of the 12th workshop on formal techniques for java-like programs - FTFJP '10*. the 12th Workshop, Maribor, Slovenia, ACM Press. https://doi.org/10.1145/1924520.1924529

Sangiorgi, D., & Walker, D. (2001). *The [pi]-calculus: A theory of mobile processes*. Cambridge, [England] ; New York, Cambridge University Press.

Scala contributors [Scala contributors]. (2018, December 1). Retrieved June 14, 2020, from https://contributors.scala-lang.org/?_ga=2.131495820.223888604.1608205572-284984504.1591719825

Scala-mungo tutorial [Scala-mungo]. (2021). Retrieved January 12, 2021, from https://aliceravier.github.io/

Scalas, A., Dardha, O., Hu, R., & Yoshida, N. (2017). A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming (P. Müller, Ed.), *74*, 24:1–24:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.24

Keywords: process calculi, session types, concurrent programming, Scala

Scalas, A., & Yoshida, N. (2016a). Lightweight Session Programming in Scala (Artifact). *Dagstuhl Artifacts Series*, *2*(1), 11:1–11:2. https://doi.org/10.4230/DARTS.2.1.11

Keywords: session types, Scala, concurrency

Scalas, A., & Yoshida, N. (2016b). *Lightweight Session Programming in Scala (Artifact)* (No. *1*). Dagstuhl, Germany, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. https://doi.org/10.4230/DARTS.2.1.11

Keywords: session types, Scala, concurrency

ScalaTest. (2009). Retrieved July 25, 2020, from https://www.scalatest.org/

Strom, R. E., & Yemini, S. (1986). Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, *SE-12*(1), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Vallecillo, A., Vasconcelos, V. T., & Ravara, A. (2006). Typing the behavior of software components using session types. *Fundamenta Informaticæ*, *73*(4), 583–598.

wartremover. (2019). *Wart remover github repository*. Retrieved September 2020, from https://github.com/wartremover/wartremover

Yoshida, N., Hu, R., Neykova, R., & Ng, N. (2013). The scribble protocol language, 22–41.

Appendix

8.3 Scalac phases

```
1  phase name   id  description
2      ----------  --  -----------
3          parser   1  parse source into ASTs, perform simple desugaring
4           namer   2  resolve names, attach symbols to named trees
5  packageobjects   3  load package objects
6           typer   4  the meat and potatoes: type the trees
7  superaccessors   5  add super accessors in traits and nested classes
8      extmethods   6  add extension methods for inline classes
9         pickler   7  serialize symbol tables
10      refchecks   8  reference/override checking, translate nested objects
11         patmat   9  translate match expressions
12        uncurry  10  uncurry, translate function values to anonymous classes
13         fields  11  synthesize accessors and fields, add bitmaps for lazy vals
14      tailcalls  12  replace tail calls by jumps
15     specialize  13  @specialized-driven class and method specialization
16  explicitouter  14  this refs to outer pointers
17        erasure  15  erase types, add interfaces for traits
18    posterasure  16  clean up erased inline classes
19     lambdalift  17  move nested functions to top level
20   constructors  18  move field definitions into constructors
21        flatten  19  eliminate inner classes
22          mixin  20  mixin composition
23        cleanup  21  platform-specific cleanups, generate reflective calls
24     delambdafy  22  remove lambdas
25            jvm  23  generate JVM bytecode
26       terminal  24  the last phase during a compilation run
```

8.4 Aliasing examples

These examples illustrate the aliasing cases described in the implementation, section 5.2.5. In these examples, if an instance is not being pointed at anymore, it is removed from the example.

Case A:

Code	**Instances**
`1 val cat = new Cat()`	`cat -> Cat@0;init`

Case B:

Code	Instances
1 `val cat = new Cat()`	`cat -> Cat@0;init`
2 `val kitten = cat`	`cat | kitten -> Cat@0;init`

Case C:

Code	Instances
1 `val cat = new Cat()`	`cat -> Cat@0;init`
2 `cat.sleep()`	`cat -> Cat@0;Asleep`
3 `cat = new Cat()`	`cat -> Cat@1;init`

Case D:

Code	Instances
1 `val cat = new Cat()`	`cat -> Cat@0;init`
2 `cat.sleep()`	`cat -> Cat@0;Asleep`
3 `val kitten = new Cat()`	`cat -> Cat@0;Asleep  kitten -> Cat@1;init`
4 `kitten.walk()`	`cat -> Cat@0;Asleep  kitten -> Cat@1;Started`
5 `cat = kitten`	`kitten | cat -> Cat@1;Started`

In case D, we see on line 5 that "cat" should now point to the same instance that "kitten" is pointing at, so "cat" is removed from Cat@0 and added to Cat@1.

8.5 User testing

8.5.1 Scala-Mungo Tutorial

The Scala-Mungo tutorial is presented on a github pages website "GitHub Pages" (2020). Here is the website as it was when user testing began:

View on GitHub

Scala-Mungo

Scala-Mungo tutorial

Scala-Mungo tutorial

Introduction

Scala-Mungo is a tool which lets you add a protocol/typestate definition to your classes. In this tutorial you will learn how to install Scala-Mungo, create a protocol for a class see an example of a program using that class; there are also exercises to try at the end.

Installing Scala-Mungo

Installing Scala

If you already have Scala, you can move on. You can download Scala here with sbt: https://www.scala-lang.org/download/ To run it on command line, you can use this tutorial: https://docs.scala-lang.org/getting-started/sbt-track/getting-started-with-scala-and-sbt-on-the-command-line.html

SBT

Copy in these lines into your build.sbt file:

```
resolvers += Resolver.bintrayRepo("aliceravier", "maven")

autoCompilerPlugins := true

addCompilerPlugin("org.me" % "scala-mungo-prototype_2.13" % "1.9")
val root = ABSOLUTE-PATH-TO-YOUR-PROJECT
scalacOptions += "-P:GetFileFromAnnotation:"+root

libraryDependencies += "org.me" % "scala-mungo-prototype_2.13" % "1.9"
```

Instead of ABSOLUTE-PATH-TO-YOUR-PROJECT, put in the absolute path to your project, i.e. the path to the build.sbt file.

It should look like this: `val root = "C:\\Year five\\Scala-Mungo-dir\\Test"`

This is used for the plugin to find the location of the protocols you will define.

Other build tools

For other build tools, see this page: https://bintray.com/aliceravier/maven/scala-mungo-prototype There is a bit at the bottom left of the page which gives code snippets to add to other build tool files. I haven't tested them so I don't know if they will work, but they should.

Fixes for problems during setup

I have been intellij and sbt for testing and have found that a lot of problems can be fixed with three techinques:

- invalidate caches and restart
- use `sbt run` instead of running the code from the editor
- put all the code into one file (classes and program to run)

Creating a protocol

A protocol consists of states that instances of the class can be in, and transitions between the states facilitated by method calls.

Specification

- The protocol must be defined inside an object which extends "ProtocolLang" and the code for the protocol must be in the main function of that object.

- States are defined with the "in" function which takes a String: the name of the state. Example: `in("init")`

- Transitions from a state can be added underneath a state definition with the "when" method which takes a String of the method signature as an argument, and the "goto" method which takes a String of the state to transition to as an argument. Example: `when("giveMoney(Int)") goto "moneyGiven"` (note that parenthesis aren't needed for the goto method's argument).

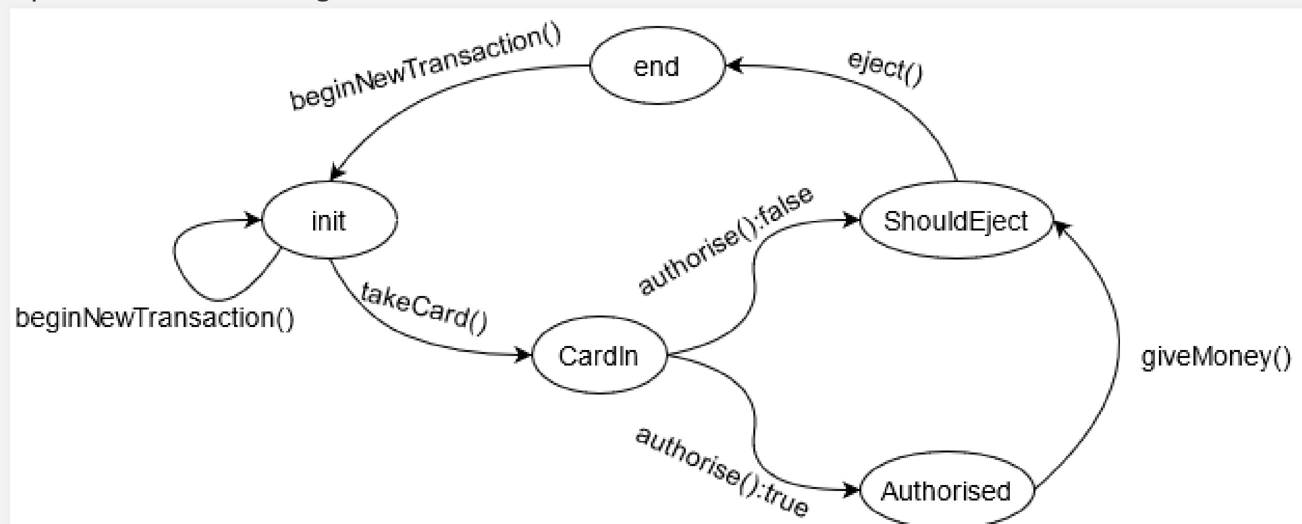
- To write a transition which depends on the return value of a method, the "at" and "or" methods are used. The "at" method specifies the return value which enables the transition and takes the return value as a String as an

argument. The “or” method lets you add a different return value and takes the String of the next state to go to for the different return value. You can keep adding “or”s to add as many return values as you want. Example: ` when(“authorise()”) goto “Authorised” at “true” or “Unauthorised” at “false” `

- Protocols must contain a unique “init” state which will be the state given to an instance when it is initialised.
- Protocols must contain a unique “end” state which indicates protocol completion, that is, at the end of a program, the object should be in the “end” state. All states must have a path of transitions between themselves and the “end” state.
- You must end your protocol with the “end()” method.
- The protocol file must be called by the name of the object which the protocol is defined it. Example: a protocol defined in the object “ATMProtocol” should have “ATMProtocol.scala” as the file name.

Example

Let’s say we want to create an ATM object which should follow a certain protocol. We want it to be able to take a card in, check if it’s authorised, and then give money if it is or eject the card if not. Then we want the transaction to be available again. We need to include an “end” state which we can place right next to the “init” state, where all the transactions should start from. We come up with the following state machine:



Now we can write our protocol. We need to extend “ProtocolLang” and create a main method:

```
import ProtocolDSL.ProtocolLang

object ATMProtocol extends ProtocolLang with App{

}
```

This needs to be in a file called “ATMProtocol.scala”.

Then we can define the states we want:

```
import ProtocolDSL.ProtocolLang

object ATMProtocol extends ProtocolLang with App{
  in("init")
  in("CardIn")
  in("Authorised")
  in("ShouldEject")
  in("end")
}
```

Then we can define the transitions for all the states:

```
import ProtocolDSL.ProtocolLang

object ATMProtocol extends ProtocolLang with App{
  in("init")
  when("takeCard()") goto "CardIn"
  when("beginNewTransaction()") goto "init"
  in("CardIn")
  when("authorise()") goto
    "Authorised" at "true" or
    "ShouldEject" at "false"
  in("Authorised")
  when("giveMoney()") goto "ShouldEject"
  in("ShouldEject")
  when("eject()") goto "end"
  in("end")
  when("beginNewTransaction()") goto "init"
 }
```

And finally we add the “end()” method at the end of the protocol:

Final version of the ATMProtocol.scala file:

```
import ProtocolDSL.ProtocolLang

object ATMProtocol extends ProtocolLang with App{
  in("init")
  when("takeCard()") goto "CardIn"
  when("beginNewTransaction()") goto "init"
  in("CardIn")
  when("authorise()") goto
    "Authorised" at "true" or
    "ShouldEject" at "false"
  in("Authorised")
  when("giveMoney()") goto "ShouldEject"
  in("ShouldEject")
  when("eject()") goto "end"
  in("end")
  when("beginNewTransaction()") goto "init"
  end()
}
```

Then we can run the protocol, which should create a file called “ATMProtocol.ser” inside a “compiledProtocols” directory in the project root. Now the plugin can find it and use it to check our ATMs are running correctly!

Make sure to run the protocol before writing code which uses it or the plugin will complain about not being able to find the protocol. In that case, comment out the @Typestate annotation and run the protocol again.

If you made a mistake in the protocol and modify it at this stage, you may need to comment out the @Typestate annotation to be able to run it again and correct the mistake in the ATMProtocol.ser file.

Using a protocol in a program

Adding a protocol to a class in the program is easy. You only need to add the @Typestate annotation to the class which should be following the protocol. The annotation takes a String of the name of the object which the protocol was defined in as an argument. In our ATM example:

```
import compilerPlugin.Typestate

@Typestate("ATMProtocol")
class ATM(){...}
```

Here is a full example program which uses the ATMProtocol defined above:

```
import compilerPlugin.Typestate

@Typestate("ATMProtocol")
class ATM {
  def takeCard(): Unit ={}

  def authorise(): Boolean ={
    var cardIsValid = false
    //code which checks if the card is valid
    cardIsValid
  }

  def eject(): Unit ={}

  def giveMoney(): Unit ={}

  def beginNewTransaction(): Unit ={}
}

object ATMtest extends App{
  val myATM = new ATM()
  myATM.takeCard()
    myATM.authorise() match {
      case true =>
        myATM.giveMoney()
      case false =>
    }
    myATM.eject()
  for(x <- 1 to 10) {
    myATM.beginNewTransaction()
    myATM.takeCard()
    myATM.authorise() match {
      case true =>
        myATM.giveMoney()
      case false =>
    }
    myATM.eject()
  }
  println("Ran the ATM program sucessfully!")
}
```

This should not cause errors to be thrown.

Try removing the “myATM.eject()” line. This should throw an error saying that the beginNewTransaction() method was called inappropriately.

If the program isn’t running and is complaining about not finding the protocol, make sure you have got a ATMProtocol.ser file inside the compiledProtocols directory. If not, comment out the @Typestate annotation and run the ATM

protocol defined in the above section.

Exercices

1: Do the ATM example explained above

2: An aliasing example

As a programmer, you want to model the flow of cash in a company. amounts of money dealt with should always be acted on in a certain way: they should be filled, have interest added to them and only then be used. Given the following state machine representation, create a protocol in Scala-Mungo for a stash of money.

You also want to have this money used by managers and a database. In the code below, a manager and a database are created which both take the same MoneyStash instance as a field.

A note on aliasing

An instance which has two ways of referring to it is called “aliased”. So in this case the MoneyStash instance is aliased. This could cause problems since the manager and database don’t know what each is doing on the other’s MoneyStash instance. If both applied interest that would be illegal from the protocol’s standpoint, but would be invisible from the standpoint of individual variables. Scala-Mungo tracks all the aliases (variable names) for a given instance so that this doesn’t happen.

Copy the code below into a file and check it works with the protocol defined above:

```
@Typestate("MoneyStashProtocol")
class MoneyStash() {
  var amountOfMoney : Float = 0
  def fill(amount : Float ) : Unit = {amountOfMoney = amount}
  def get() : Float = amountOfMoney
  def applyInterest(interest_rate : Float) : Unit = {
    amountOfMoney = amountOfMoney * interest_rate;
  }
}

class DataStorage() {
  var money : MoneyStash = null;
  def setMoney(m : MoneyStash) : Unit = {money = m}
  def store() : Unit = {
    var amount = money.get()
    println(amount)
    // write to DB
  }
}

class SalaryManager() {
  var money : MoneyStash = null;
  def setMoney(m : MoneyStash) : Unit = {
    money = m
  }
  def addSalary(amount: Float) : Unit = {
    money.fill(amount)
    money.applyInterest(1.02f)
  }
}

object Demonstration extends App {
  val salary = new MoneyStash
  val manager = new SalaryManager
  val storage = new DataStorage

  manager.setMoney(salary)
  storage.setMoney(salary)

  manager.addSalary(5000)
  storage.store()
}
```

Now try adding a “money.applyInterest(1.02f)” line above the “var amount = money.get()” line in the DataStorage class. This should cause an error when run.

3: Create a protocol from a written specification

Now write a protocol for a cat, or any other animal of your choice.

- The animal must be able to complete any number of walk()-slow()-stop()-startAgain() cycles, from init.
- It must also be able to complete any number of walk()-run()-slow()-stop()-startAgain() cycles, from init.
- From init, when calling the sleep() method, it might fall asleep (return true), in which case it should then be able to call awaken() and then startAgain(). It should be able to repeat this infinitely.
- From init, it might also call the sleep() method which would return “false”, in which case it shouldn’t change anything to the state. It can call sleep():false an infinite number of times.

Remember that protocols must contain a unique “end” state which indicates protocol completion, that is, at the end of a program, the object should be in the “end” state. All states must have a path of transitions between themselves and the “end” state.

Once you have written this protocol, run it and then write a program which uses your animal class and does not error. Then write a program which does error. You may use the class below or write your own one:

```
class Cat{
  def walk(): Unit ={

  }
  def slow(): Unit ={

  }
  def stop(): Unit ={

  }
  def run(): Unit ={

  }
  def sleep():Boolean ={
    true
  }
  def awaken(): Unit ={

  }
  def startOver(): Unit ={

  }
}
```

8.5.2 Questions

- How was the set-up step?
- How difficult were the exercises for you, did the error messages help?
- Approximately how much time did the exercises take?
- Did you find any bugs, what were they?
- What are things which could be improved?
- Any other comments?

8.5.3 Answers

User tester A

- *How was the set-up step?*

  It took me a while to understand what to do at the beginning since I am new to Scala and SBT.

- *How difficult were the exercises for you, did the error messages help?*

  The exercises were easy. The error messages seem ok. When an invalid transition is found, the set of states could be represented with {} instead of TreeSet() though.

- *Approximately how much time did the exercises take?*

  I am not sure, probably half an hour, including the setup. Without the setup, maybe 15 minutes.

- *Did you find any bugs, what were they?*

  In the MoneyStash example, I can call "storage.store()" twice. In the way I defined the protocol, I assume that after the "get" call, the "end" state is reached and no other methods should be called.

- *What are things which could be improved?*

  The plugin's name could be more specific, like "scalaMungoPlugin", instead of "compilerPlugin".

  Additionally, the fact that one needs to run the protocol and comment out the @Typestate annotations to then make the compilation of the app work does not seem to be very developer friendly.

- Any other comments?

  Good job!

User tester B

- *How was the set-up step?*

  A bit confusing; it also seems targeted to windows, and intellij. Perhaps a step-by-step would be helpful, i.e. create scala project, modify the build, etc. And the versions of libraries/tools needed.

- *How difficult were the exercises for you, did the error messages help?*

  The exercises were not difficult and were good at illustrating how to use the tool. However, when trying to run the annotated program I kept getting the following error: `NoSuchMethodError:scala.reflect.internal.AnnotationInfos$AnnotationInfo$.unapply(Lscala/reflect/internal/AnnotationInfosAnnotationInfo;)Lscala/Option;`

- *Approximately how much time did the exercises take?*

  About an hour, but a good chunk of that was spent on set-up/trying to get rid of the above error.

- *Did you find any bugs, what were they?*

- *What are things which could be improved?*

- *Any other comments?*

  Better instructions for installing and running the tool would be helpful.

User tester C

- *How was the set-up step?*

  i) The setup was complicated by the fact that it seemed that a specific Scala version was needed to run the example. The default choice from SBT did not work.

  ii) Due to my inexperience with Scala getting the directory structure right took some tries, but that has nothing to do with the tool.

  iii) Having to execute the protocols before adding them to the @Typestate annotation complicated the process of using them. Especially since they have to be chosen explicitly in the menu that sbt shows on "sbt run".

- *How difficult were the exercises for you, did the error messages help?*

  The exercises weren't very difficult to me, as I have experience with such protocols (and similar tools).

  ii) The error messages were helpful, and caught issues with my programs as I was writing them. For example in the final exercise the error messages told me that calling awaken() directly after sleep() was not allowed, hence i realized that I had to wrap it in an match-statement.

iii) The stack trace that accompanied the error messages were not useful when using the tool.

iv) I was a bit confused by the “cannot reach end” error message, even though in retrospect it clearly states the problem.

v) There is some implementation detail showing in the error messages such as “States(init, sleeping)” rather than “the states ’init’ and ’sleeping’”.

- *Approximately how much time did the exercises take?*

  About an hour, including writing code, compilation/type checking and setup.

- *Did you find any bugs, what were they?*

  i) The initial ATM example did not work directly. I had to remove the outer loop to get the type checker to accept it.

  ii) I’m not sure if it’s a bug or just well-defined Scala parsing, but there are some inconsistencies around where line breaks are required in the protocol definitions. ‘when(”walk()”) goto(”walking”)‘ is allowed whereas ‘in(”end”) end()‘ is not

- What are things which could be improved?

  The double compilation/running step is the biggest annoyance I think. Maybe the @Typestate annotation could receive the protocol as an argument instead, so that the protocol weren’t in a different file? I don’t know if this is at all how the embedded language works, but it would be easier to work with then.

- *Any other comments?*

  i) It felt very natural to write the protocols in the defined language. Having the language extended with the syntax for protocols, rather than writing them all as long strings, makes them easier to both read and write.

  ii) Some error messages could be better such as: “Instance of type ATM with aliases TreeSet(myATM) did not necessarily reach its end state. At the end of the program it was in state(s) TreeSet(init, end)”. To me it’s not clear what TreeSet(init, end) means, and I have to resort to guessing what the type system is telling me.

User tester D

- *How was the set-up step?*

  Difficult. Although it was mostly the fault of sbt and me since my jdk version was wrong.

- *How difficult were the exercises for you, did the error messages help?*

The exercises were not difficult except the setup and the error messages helped a lot. I got the following errors related to the protocol specification:

- "compilerPlugin,unendedProtocolException: Instance of type ATM with aliases TreeSet(myATM) did not necessarily react its end state. At the end of the program it was in state(s) TreeSet(init, end)". This was hard to read since the end state was part of the state TreeSet hence it did not understand what the issue was.
- "Did you mean goto" when("get()") go "end" this was very helpful
- "compilerPlugin.badlyDefinedProtocolException: methods set(...) are not a subset of methods set(...)" also very helpful
- "compilerPlugin.protocolViolatedException: Invalid transition in instance with ...". This was great and nice that there were line numbers!
- "Invalid transition in instance .. from state Treeset(end) with method startAgain(). Possible methods to use in this state are: startAgain()" hard to see that I was missing parenthesis in the protocol. I had written startAgain goto "init"in the protocol.

• *Approximately how much time did the exercises take?*

• *Did you find any bugs, what were they?*

Maybe. I specified "startAgain" in the protocol without parenthesis and I did not get a compile time error saying that what I specified to be a method transition is wrong since "startAgain" is not a method call. (missing some validation on transition method names in protocols?).

• *What are things which could be improved?*

- Error checking in the editor. I'm not sure if this is already implemented if I had used Intellij for the exercises, if not, then I think it would be an amazing addition.
- Some error messages were hard to read.
- It would be cool to have some method transitions that could be called anytime without having to explicitly specify them for every state. For example by allowing an object to be in multiple states at the same time.

• *Any other comments?*

Nice with some aliasing and overall I think its a good plugin. The error messages are for the most part very helpful and its nice to write protocols and examples in a "real language" instead of a calculus for once.